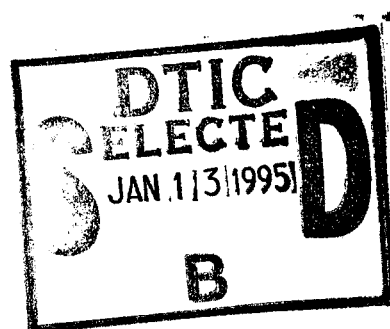# NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SYNTHETIC ENVIRONMENTS
FOR
C3 OPERATIONS

by

John M. Young

September, 1994

Thesis Advisor: Morris R. Driels

Approved for public release; distribution is unlimited.

19950112 002

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1994.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Synthetic Environments For C3 Operations	5. FUNDING NUMBERS
6. AUTHOR(S) John M. Young	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE *A

13. ABSTRACT

Modeling, simulation, and display of information and situations have helped people make decisions since the first diagram was drawn in the mud. Today, computer hardware and software developments have advanced to allow very sophisticated and nearly real-time displays. The introduction of virtual reality simulations into the C3 environment can significantly improve the amount and display quality of information. World Tool Kit developed by Sense8 Corporation has been used to produce a simulation. The scenario has two opposing battle groups closing the distance of ocean between them, to demonstrate some of the potential advantages of this new and mostly untapped potential. The focus is on introduction of the technology into the C3 environment and will deal with some of the fundamental advantages and difficulties.

14. SUBJECT TERMS Synthetic Environments For C3 Operations			15. NUMBER OF PAGES * 94
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

Approved for public release; distribution is unlimited.

Synthetic Enviroments
For
C3 Operations

by

John M. Young
Lieutenant, United States Navy
B.S., United States Naval Academy, 1987

Submitted in partial fulfillment
of the requirements for the degree of

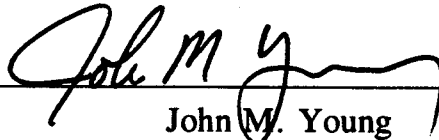MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1994
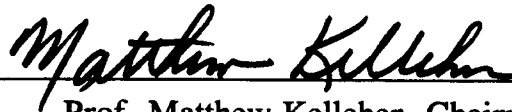
Author: _____
John M. Young

Approved by: _____
Prof. Morris R. Driels, Thesis Advisor

_____
Prof. Matthew Kelleher, Chairman
Department of Mechanical Engineering

ABSTRACT

Modeling, simulation, and display of information and situations have helped people make decisions since the first diagram was drawn in the mud. Today, computer hardware and software developments have advanced to allow very sophisticated and nearly real-time displays. The introduction of virtual reality simulations into the C3 environment can significantly improve the amount and display quality of information. World Tool Kit developed by Sense8 Corporation has been used to produce a simulation. The scenario has two opposing battle groups closing the distance of ocean between them, to demonstrate some of the potential advantages of this new and mostly untapped potential. The focus is on introduction of the technology into the C3 environment and will deal with some of the fundamental advantages and difficulties.

ACKNOWLEDGMENT

TABLE OF CONTENTS

I. INTRODUCTION

A. PROBLEM DEFINITION:

People and the their decisions shape history. Many decisions are made after long periods of deliberation, sometimes years; others are made immediately. The importance of a decision is not however necessarily proportional to the amount of time of deliberation. In today's highly technical society and with almost instantaneously communication around the world, the assessment time for many critical decisions is much less . Therefore it is crucial that decision-makers be given clear and concise information rich presentations.

The importance of business decisions cannot be over emphasized, because jobs, livelihoods and quality of life are at stake. However in the military, it is even more important that the best decisions be obtained, because people die unnecessarily when bad decisions are made. Combat Information Centers and War rooms are where information and data converge. It is vital that the leaders and strategist be able to have a clear and complete picture. Information is collected from a myriad of sources, but information that is known is not useful unless it is presented in a timely and usable form to the right people. The problem for any decision maker is how to present the most information clearly and in the most useful form, consolidated and converting much of the information that is spread out or buried in unusable or illegible formats.

B. BACKGROUND:

Computers and computer technology is evolving at an incredible rate. As with any area of study, new terms are created to define previously non-existent or undiscovered

ideas and developments. Much of the terminology associated with computers is new and not generally common knowledge, therefore some definitions will be introduced.

"VIRTUAL REALITY is a term coined by Jaron Lanier, founder of VPL Laboratories to distinguish between the immersion in digital worlds and traditional computer simulations"[Ref. 1, p. xv]. Traditional simulation is a standard screen display while immersion is an open ended reference to sensory stimulation such as stereoscopic goggles, motion sensitive clothing, etc.

CYBERSPACE defines the alternate world created in virtual reality.

MOUSE is a two dimensional input device.

SPACEBALL is an input device that allows six dimensional input.

UNIVERSE is the container for the objects in the virtual world.

SYNTHETIC ENVIRONMENT is an inclusive model.

C.   TOOLS:

Computer modeling can be extremely labor intensive. If a new computer program had to be written for each application the manpower and time lost would out weigh the added benefits. There is software available that can be adapted for a vast number of applications. Choosing an appropriate software involves finding one with the right attributes. The software must be flexible. It must be adaptable to a wide range of applications. If the program is too difficult to learn or modify then even a very powerful program will not be used or used to its fullest extent. Speed a major factor. An application should be able to be built quickly, possibly through sensor input or automatically, but ideally as close to real-time as possible. Lastly, the software must provide adequate features to provide a powerful display that is truly an asset to the decision-making process and not just window dressing.

World Tool Kit (WTK) developed by Sense8 Corporation was chosen for its combination of these attributes. Autodesk has a similar product, the Cyberspace Development Kit, that was not evaluated and could be an area of future consideration. This type of software is in its infancy with poor documentation a few references. The review of WTK in Chapter II coupled with the application in Appendix A show some possible uses, but is in no way exhaustive.

D. OBJECTIVES:

The purpose of this thesis is to show that computer simulation modeling can improve the information available to decision-makers. For this application, two battle groups consisting of airplanes, helicopters, and ships are represented by symbols. A three dimensional grid 300 x 300, with a scale factor of one unit equals one nautical mile, models a section of open sea and air space and will be used as the forum. A single scenario will be used to demonstrate the ability for planning and also real-time decision making. It will be assumed that during the real-time phase, assets are moved according to the latest intelligence or sensor input. To demonstrate the purpose, WTK will be introduced and utilized to develop and build the application. The expectation is to prove that much of the information now spread out on status boards and two dimensional displays can be consolidated effectively using WTK. This thesis is to validate the concept of introducing virtual reality software into the decision making environment. It should not be considered as the full utilization of this technology. Many of the features available were not needed to meet the objective and greater utilization should be considered for future research.

II. WORLD TOOL KIT REVIEW

A. OVERVIEW:

World tool kit (WTK) is a set of subroutines written in the C programming language designed to allow the user to create 3-D simulations and models. The power of WTK is that it allows users with a basic knowledge of C programming to produce simulations and models that without WTK would require extensive computer knowledge and advanced programming techniques.

WTK is available on several computer platforms including personal computers and workstations. There is a user's group available that provides 3-D models and is also used as a forum for an exchange of ideas and techniques. The user's group was not contacted nor were any models down-loaded from their library for this thesis, therefore a determination of its usefulness can not be addressed.

C programming and WTK are object-oriented. This means that properties and attributes can be inherited or passed on. It can be very confusing because in WTK there is the graphical object class of subroutines and other classes which are referred to as objects. The graphical object class, also referred to as the object class, is a class of functions that is used to generate or manipulate graphical objects in a scenario. However, other classes are also referred to as objects. This is a reference to the structure of data or properties that can be inherited or passed on. For example, an input device such as a mouse has a data structure associated with it that produces and stores an absolute record. This can be inherited by another input device, therefore sensors are referred to as sensor objects.

There are over 400 subroutines in the WTK library. These are broken down into classes of functions, which include universe, graphical object, window, sensor and more. Each class of function has its own set of subroutines identified by a handle. A handle is defined as a previously defined pointer to a structure of the type defined by the class of the function called. For example **WTobject_new** is the handle for the function that introduces a graphical object into a simulation. The actual C programming code would be **WTobject *WTobject_new.** Most classes require that a handle be associated with the function call as a pointer to the stored data returned from the function.

WTK is a visual simulation media, in which graphical objects and their realism are specifically important to the simulation and modelling. Computer aided design (CAD) programs such as Autocad by Autodesk are particularly useful for producing the graphical objects used in WTK simulations. In addition to being able to import graphical objects, WTK has function calls available to produce basic geometric objects like spheres, cones and boxes. WTK is compatible with DXF format as well as its indigenous format Neutral File Format (NFF).

Surfaces of a graphical object in WTK can also be textured. Texturing is the attaching of a "covering" to a graphical object to give it a more realistic appearance. An example might be texturing a terrain surface with a scanned photo of grass or texturing a teapot with the image of brass.

The preceding introduction is by no means exhaustive and to further demonstrate the capabilities of WTK it is necessary to briefly discuss the classes of functions available. In the sections to follow a general scope of each class of function will be presented to provide a better understanding of WTK. Chapter III will introduce the simulation example and will describe functions of each class in detail as well as a process for building a scenario.

B. UNIVERSE CLASS:

As implied by the name, the universe class deals with the enclosure that encases the simulation. This is a stationary graphical object that is mandatory. The universe can be created or loaded. Many of the universe functions such as **WTuniverse_new** and **WTuniverse_load** functions do not have handles because there can only be one universe at a time. However, other universe functions such as **WTuniverse_getlights** require handles because there can be more than one. **WTuniverse_new** or **WTuniverse_load** must be one of the first if not the first statements in the main program. These functions create the container that will house the simulation. **WTuniverse_getlights** returns a pointer to the first light from a list of all lights in the simulation.

The universe class functions also are used to define and control the repetitive simulation loop illustrated in Figure 2.1.[Ref. 2, p. 2-8]

WTuniverse_go()
to enter simulation loop

Sensors are read.

The universe's action function is called.

Objects are updated with sensor input.

Graphical objects perform tasks.

The universe is rendered.

WTuniverse_stop()
to exit simulation loop

Figure 2.1

7
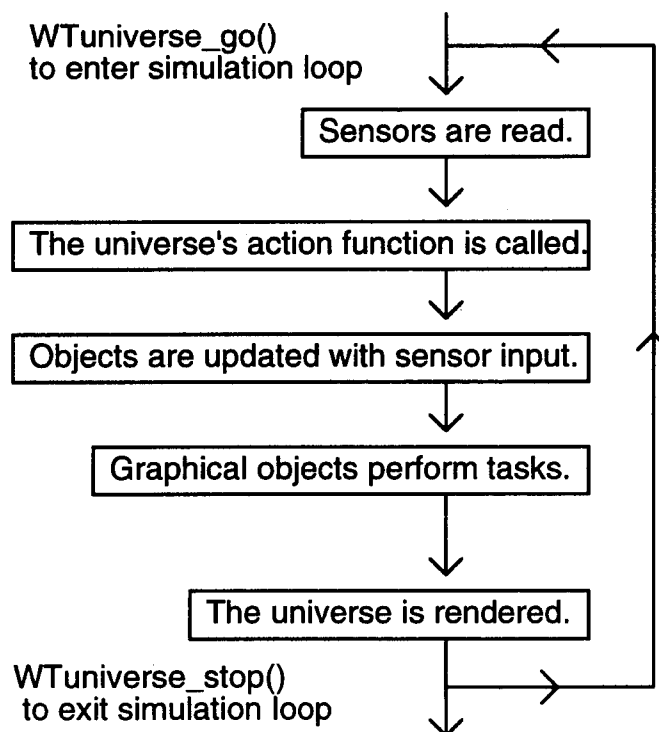
WTK is not just a static simulation and can receive input, move objects or viewpoints and present an updated visual rendering. More precisely, sensors are read, a user defined universe action function is called, objects are updated with sensor information, graphical objects perform defined tasks, the universe is rendered and then the loop is repeated.   The order of the universe action function, object update from sensors, and tasks by graphical objects can be interchanged into any order as defined by the user, to allow great flexibility for the application programmer.

C. OBJECT CLASS:

The graphical object class is the basic unit of WTK simulation. Objects can be manipulated in a multitude of ways, including creating networks of hierarchy, sensor attachment, tasking and user defined data structures can be assigned and associated with them. This flexibility is the heart of WTK's effectiveness and versatility.

WTK will import both NFF and DXF as well as indigenously producing basic shapes as noted in Section 2A. Some care must taken when creating and loading new graphical objects. When rendering graphical objects, WTK can accept or reject the backface of the graphical object, depending on how it is designed or the defining settings. If backfaces are rejected then the graphical object may be seen from one side only. For instance, if a wall is created with backface rejection and the viewpoint is from in front of the wall it will be rendered and appear normally. If the viewpoint is on the other side of the wall it will not be rendered and will not appear in the simulation from that perspective. This is also a problem when a viewpoint is from within a polygon with backface rejection. Once inside, the polygon will disappear from that viewpoint. Coplanar polygons also can lead to difficulties if the user is not careful. These hurdles are explained in the user's

8

manual provided with the software and it is recommended to review this section before building an application.

In any graphical application the reference frame system is critical. WTK uses the right hand rule for defining reference frames and provides both world and user defined local reference frames. The world reference frame has the screen as the X-Y plane with positive X to the right and positive Y going down. The Z axis is perpendicular to the X-Y plane with positive Z going into the screen. For the local frame the X and Y axes generally coincident with the longest and next longest dimensions respectively. These axis orientations are illustrated in Figure 2.2.

World coordinate system   Local coordinate system
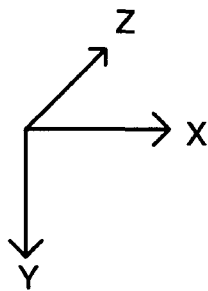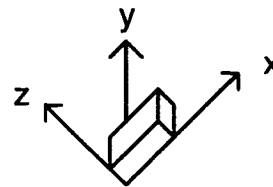
Figure 2.2

Complete representation of a graphical objects placement requires position and orientation. Positioning is fully described by X,Y, and Z coordinates, while orientation is handled with quaternion representation. A quaternion is a 4-D representation that avoids the singularities of 3-D orientations. The quaternion is a vector in 3-D and a rotation about that vector.

D. POLYGON CLASS:

The polygon class is very similar to the graphical object class, but not as versatile. The polygon class offers the ability to access vertices directly, which can be a very powerful advantage. Another useful feature of the polygon class is that each polygon is assigned a unique ID number. Intersection testing is also available in this class. Overall the polygon class can allow the user to perform graphical representations quickly and provides for easier editing of the objects. Many of the features provided with the graphical object class are however not available.

E. VERTEX CLASS:

The Vertex class is a set of functions to access information about existing vertices. It does contain a limited capability using the **WTvertex_new** command to define vertices in restricted application, but is generally an informational class of functions.

F. SENSOR OBJECT:

Input is the primary function of the sensor class. These subroutines allow the user to interact with the simulation and is key to realistic simulation. WTK supports most advanced computer input devices such as the mouse, the spaceball and many others. Sensors can be used to drive graphical object motion and viewpoints as well as actions, events and animations.

A WTK sensor handle must be created to utilize a sensor. Once this has been done WTK manages the input through the subroutines automatically. This permits the user to switch inputs or to use devices that have relative and absolute coordinates frames interchangeably. The raw data input is converted and passed to the simulation in usable form without interaction or manipulation by the user.

G. LIGHT OBJECT:

Lighting is essential to any optical rendering. The background or ambient lighting controls the overall illumination of the simulation while directed lighting shows and accentuates the contours of the objects. WTK has both ambient and directed lighting. Light intensity is scaled from 0.0 (black) to 1.0 (maximum).

Ambient lighting is default set to 0.4 and can be adjusted at any time before or during the scenario. Directed lights can be added without limit. Directed lighting and objects do not act exactly as in reality. Polygons and graphical objects do not cast shadows. Lighting is not blocked by graphical objects in it's path nor does it attenuate with distance. Shading is recomputed for each rendering of the simulation and it takes the same amount of time to compute shading for an intensity of 1.0 as it does for 0.0. Lights can be turned on or off at any time during the scenario.

H. VIEWPOINT CLASS:

Viewpoints and the manipulation of their position and orientation can add significantly to the realism of a simulation. WTK can render both monoscopic and stereoscopic viewpoints. The viewpoint consists of several defined parameters that determine how a frame is rendered. The understanding of these parameters is paramount to maximizing the advantages of WTK.

These parameters are position, orientation, direction, angle, aspect ratio, hither clipping plane value, parallax, convergence and convergence distance. It is useful to think of viewpoint as a camera. Position, orientation and direction are exactly as their names suggest, geographical location, rotational alignment, and vector bearing of the camera. The angle refers to angular width in radians from the camera position to the center and

11

right edge of the view area. The hither clipping plane value is the distance from the camera to a plane in which any object between the camera and this plane is not rendered. Parallax is the distance between right and left views in a stereoscopic viewing. Convergence is the horizontal offset in pixels between right and left views. Convergence distance is an additional parameter available on some platforms that support asymmetric projection.

I. ANIMATION CLASS:

Animation class functions are used to allow 3-D objects to dynamically change. This class might be used for morphing objects, or assembling objects. Animations are the construction of a sequence of objects from individual models that may or may not be similar.

J. PATH CLASS:

Paths are collections of positions and orientations. Viewpoints and graphical objects can be attached to paths and moved and rotated along that path. Each combination of a position and an orientation is called a node. Paths are edited by inserting or deleting nodes. Speed and direction along a path can also be modified. A significant path editing feature is the ability to smooth a path by interpolating between existing nodes. Interpolation method can be selected from three different types, allowing design flexibility.

K. TERRAIN CLASS:

The terrain class provides an intelligent and user friendly method of modeling terrain. Terrain can be created as flat, random height or data driven. The general concept utilized by the terrain functions can be theorized as taking a flat two dimensional area or

grid, determining the height of each position on the grid and connecting the three dimensional positions to obtain a terrain object. The height for each position on the grid can be zero as in a flat object, randomly generated, or read from a data file. The more grid points provided the "smoother" the terrain will appear.

L. PORTAL CLASS:

A portal is a connection between two universes. Since only one universe can be displayed at a time, portals provide a way to change a universe without starting over. Portals also allow models to be built in a modular manner, and can improve rendering efficiency.

M. TEXTURE CLASS:

The texture class adds realism. A covering can be applied to a graphical object to improve it's visual appearance. This is a significant improvement over straight colorization in that it allows graphical object to obtain almost photo-realistic detail, however simulations are slowed by texturing. Textures can be scanned in from photographs or produced by computer paint programs. In addition, a library of textures is provided with the software and more are available through the user group.

N. WINDOW CLASS:

The window class is hardware specific and is not available on all systems. The window class allows the user to position, resize and add display windows on the screen. This feature is very useful when different viewpoints are required simultaneously.

O. MATH LIBRARY:

The math library is a set of WTK functions that assist in mathematical manipulations and conversions. Many of the WTK data structures can be handled by the user without having to manipulate the individual members of the data structure. For example, adding two positions would involve individually adding the X, Y, and Z components, but it can be handled with one line utilizing a math library function. This class significantly reduces the management of the mathematical aspects of building and running a simulation.

P. DEFINED CONSTANTS:

WTK provides a set of defined constants facilitate simulation construction and manipulation. These include logical constants, indexing constants and more. It is recommended that the user review this section in the user's guide before building a simulation.

III. BUILDING AN APPLICATION

A. C PROGRAMMING FUNDAMENTALS:

All WTK applications are computer programs written in the C programming language. A short introduction to three basic C programming topics or techniques will make it easier to build and understand an application in WTK. These three topics are the standard form, pre-processing directives, and pointer variables. These three topics are not exclusively the only C programming knowledge required to create an application, and therefore further review may need to be conducted as individually required.

C programs have basic components. A standard form is depicted in Figure 3.1.

```
header statements - one or more-
main()
{
C programming statements;
}
other_functions()
{
C programming statements;
}
```

Figure 3.1

The header statements include required preparations that must be completed before the main program can run. Following the header section is a function call for **main()**, note there is no punctuation on this line. After the function call line there is an opening bracket followed by a series of C programming statements. These statements must all end with the proper punctuation required, most often a semicolon. A closing bracket is mandatory after the last statement of the function **main()**. Functions other than the main function

15

are optional but should be placed after the closing bracket for main and in the same format. In Figure 3.1 the function call for **other_functions**() is given as a example.

C is a very powerful language because it is very generic. C programming statements can be run on almost any computer available today. C accomplishes this by including generic function calls to compensate for the incompatibilities or differences in machines. For example, to print to the screen, regardless of the type of machine you are on you would write the statement:

printf("Print this to the screen");

The programmer must include the function **printf**(). It must either be written to interface properly with the particular equipment being used or the computer must have access to one that has already been written for the equipment. Available in separate files with most compilers are commonly used functions, such as standard input and output functions, common mathematical functions, and more. These two examples are usually found in files named **stdio.h** and **math.h** respectively.

The pound sign (#) indicates a pre-processing directive, which means that the action following it will be completed when the program is compiled.

#include <stdio.h>

The statement above tells the compiler to replace the **#include <stdio.h>** statement with the contents of the file **stdio.h**.

All variables must be declared prior to being used. There are two types of variables, global and local. Global variables can be thought of as a single entity while local variables can be thought of as distinct and separate variables with the same name in distinct and separate subroutines and functions. Globals are available to all functions all the time and are usually declared in the header section and preceded by the term **static**. Local variables are only available in the functions that they are declared in. Changing the

16

value of a global variable changes the entity, so the function is changed everywhere it occurs. Changing the value of a local variable in a function only effects that particular variable and all other distinct and separate variables with the same name in other functions remain unchanged.

The last important foundational C programming technique for writing an application is pointing to variables by address, referred to as pointers. Assume a floating point notational variable **dummy** has been declared. This can be accomplished with the line below.

float dummy;

The computer allocates a memory address of appropriate length to store the value of a floating point number for **dummy**. Then the a value such as 5.2 can be assigned to **dummy** with the following line of code.

dummy = 5.2;

The computer goes to memory address previously allocated for **dummy** when it was declared and inserts the floating point value **5.2**. A powerful feature of C programming is that the value of **dummy** can be determined by accessing this memory location directly. The value can also be changed by "shoving" a new value into this memory location.

First define an integer variable **add_dummy** to store the memory address of **dummy**. This is accomplished as follows.

float *add_dummy;

The "*" can be roughly translated as " the contents of". The previous statement establishes an integer variable known as a pointer, **add_dummy**. The contents of the memory location pointed to by the integer variable **add_dummy** will be a floating point number. To assign **add_dummy** the integer memory location value of **dummy** use the statement below.

<div align="center">

add_dummy = &dummy;

</div>

The "&" can be roughly translated as the "the address of", and therefore the above statement could be interpreted as, "**add_dummy** equals the address of **dummy**."

To assign a new value to **dummy**, dereference the memory location using the pointer as shown.

<div align="center">

*add_dummy = 6.0;

</div>

The above statement can be translated as, " The contents of memory location **add_dummy** equals **6.0**. Pointers are extremely important and are used very frequently in WTK.

B. CORE APPLICATION:

Separately both WTK and C programming have been introduced. Now it is time to merge the two and to begin to create an application. The programs that follow should run if typed verbatim, provided the WTK libraries are setup properly, and the users has a Microsoft compatible mouse and Spaceball technologies spaceball connected. The most basic WTK application simply creates a new universe and allows the user to end the application. A sample program is provided as Figure 3.2. Comments in C programs are in the format **/*comment goes here*/** and are included in the programs that follow for each new line of code introduced. These are ignored by the compiler. The program in Figure 3.2 is complete, however it is entirely devoid of any rendering. The reason this program is significant is because it is a core or the minimum statements for a WTK application. More complex applications include or modify the statements of this core.

The first three statements of the core program are pre-processing directives.

```
#include<stdio.h>
#include"wt.p"
#include"wt.h"
```

```
                /* CORE WTK APPLICATION */

/* HEADER SECTION */

/* Pre-processing directives -"Stdio.h" file contains the printf() function. */
#include<stdio.h>

/* Pre-processing directives -WTK functions and constants are defined in the
   files "wt.h" and "wt.p". */
#include"wt.h"
#include"wt.p"

/* Global declaration of a pointer to a WTsensor type structure "sensor". */
            static WTsensor *sensor;

/* Global declaration of the actions function. */
            static void actions();

/* MAIN PROGRAM -A basic component of a C program. */

            main()
  {

/* WTuniverse_new function call, should always be 1st WTK function call, but after local variable declarations. */
                    WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Assigns the return of WTspaceball_new() to the structure pointed to by sensor. */
                    sensor = WTspaceball_new(COM1);

/* Prepares for the simulation */
                    WTuniverse_ready();

/* Sets the actions function for inclusion in simulation loop. */
                    WTuniverse_setactions(actions)

/* Starts the simulation loop. */
                    WTuniverse_go();

/* Deletes the universe- simulation loop must be exited to reach this statement */
                    WTuniverse_delete();
                    }

        /* UNIVERSE ACTIONS */


        static void actions()

        {

/* Tests for button 1 on the spaceball through the WTK function call WTsensor_getmiscdata(). */

                    if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1)
                    {

/* Prints "BUTTON 1 ACCEPTED" utilizing printf() function in file stdio.h. */
                            printf("BUTTON 1 ACCEPTED \n");

/* Halts the simulation loop and exits to the statement following WTuniverse_go(). */
                            WTuniverse_stop();
                    }

        }
```

Figure 3.2

Each of these statements is replaced by the contents of the files indicated. **Stdio.h** contains many of the standard input and output files for programming such as print functions. The files **wt.p** and **wt.h** contain the WTK libraries. The next two lines are global declarations.

<div align="center">

static WTsensor *sensor;

</div>

Sensor is a pointer to a **WTsensor** type structure and is globally defined.

<div align="center">

static void actions();

</div>

The **actions()** function is defined globally and is analogous to **other functions()** in Figure 3.1. The main function call follows:

<div align="center">

main()

</div>

No local variables are utilized in this program and therefore the new universe function call is next. It is important to note that local variables must be defined before the new universe call, but that the new universe call should be the first WTK function call.

<div align="center">

WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT);

</div>

The parameters inside the **WTuniverse_new()** function are defined WTK constants. **Sensor** is then assigned utilizing the new spaceball function call.

<div align="center">

sensor = WTspaceball_new(COM1);

</div>

The parameter **COM1** defines the serial port connection of the spaceball.

<div align="center">

WTuniverse_ready();

</div>

The universe ready function prepares the simulation by completing all the necessary computations for rendering. WTK allows the user to include an externally written function in the WTK simulation loop.

<div align="center">

WTuniverse_setactions(actions);

</div>

The setactions function declares this function so that WTK will include it in the loop.

<div align="center">

WTuniverse_go();

</div>

The go function starts the simulation loop. Upon exiting this loop the following statement deletes the universe in preparation for another simulation.

WTuniverse_delete();

The main function ends at this point, however the **actions()** function is defined in the remaining statements. For this program the **actions()** function is simply to allow the user to end the simulation. Otherwise, the simulation loop in Figure 2.1 will be executed indefinitely. The first statement declares the function.

static void actions()

Next is a logical "if" statement that retrieves the contents of the sensor through the first argument in the logical statement and determines if "Button 1" of the spaceball has been depressed.

if (WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1)

If true, then the **printf()** function defined in **stdio.h** is called to print "BUTTON 1 ACCEPTED".

printf("BUTTON 1 ACCEPTED \n");

Finally, the simulation stop function is called and the scenario is terminated.

WTuniverse_stop();

C. ADDING OBJECTS:

The end result of the core program is a stationary viewpoint looking out into an empty universe. The default viewpoint is at the position (0,0,0) looking down the positive Z axis, directly into the screen. With only a few modifications a block can be added to the core application. These lines along with appropriate comments have been added to the core program as Figure 3.3. Note that the comments included in Figure 3.2 have been

21

```
            /* GRAPHICAL OBJECTS SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

        static WTsensor *sensor=NULL;
        static void actions();

/* Global declaration of a pointer to a WTobject type structure "block". */
        static WTobject *block;

/* MAIN PROGRAM */

    main()
    {

/* Local declaration of the variable "pos" a WTp3 type structure. */
        WTp3 pos;

/* Local declaration of the variable "ornt" a WTq type structure. */
        WTq ornt;

/* Establishes "pos" = (0,0,10) */
                    pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;

/* Establishes "ornt" = (0,0,0,1) */
                    ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;

                    WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Creates a graphical object and assigns "block" to point to it's location in memory. */
        block = WTobject_newblock(1, 2, 3, FALSE, FALSE);

/* Positions the graphical object pointed to by "block" at (0,0,10) */
        WTobject_setposition(block,pos);

/* Orients the graphical object pointed to by "block" with the quaternion "ornt". */
        WTobject_setorientation(block,ornt);

                    sensor = WTspaceball_new(COM1);
                    WTuniverse_ready();
                    WTuniverse_setactions(actions);
                    WTuniverse_go();

                    WTuniverse_delete();
    }

        /* UNIVERSE ACTIONS */

        static void actions()
        {

                if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                {
        printf("BUTTON 1 ACCEPTED \n");
                        WTuniverse_stop();
        }
        }
```

Figure 3.3

removed to highlight the changes. To add the graphical object, first any variables used must defined, then the graphical object is created, positioned, and oriented. If position and orientation are not given the default values of position (0,0,0) and orientation aligned with the world axis are used.

The best way to understand the new code is to detail each change separately. The first new line is the global declaration of the pointer **block** which points to a **WTobject** type structure.

<p align="center">**static WTobject *block;**</p>

It is essential to declare every variable before it is be used. The graphical objects and sensors are best defined globally to allow them to be known or defined in all functions and subroutines. In contrast, if the objects are defined inside of the main function they will not be known in the action function or any other function, and this will result in an application that will not compile. The first line in the function **main()** is the local declaration of a **WTp3** type variable **pos**.

<p align="center">**WTp3 pos;**</p>

The **WTp3** structure is defined in one of the files included as the pre-processing directives either, **wt.p** or **wt.h**. **Pos** is going to be the position of the object. The next new line is very similar in form, but very different in function.

<p align="center">**WTq ornt;**</p>

The orientation is defined by a quaternion, a three dimensional vector and a rotation about that vector. This line defines the variable **ornt** as a quaternion structure.

Since the viewpoint is at the default (0,0,0) and pointed down the positive Z axis, the object must be placed somewhere in the field of view if it is to be rendered in this example. If a unit cube were placed at (0,0,0) the viewpoint would be inside the cube and two scenarios are possible. One, if the cube is constructed with backfaces rejected the

cube will be there but not rendered since the view will only be of backfaces. Two, if backfaces are not rejected the entire view will be filled with the inside of the cube wall. To avoid these scenarios the object should be placed in the positive Z direction. In Figure 3.3 a position on the X and Y axis ten units down the Z axis was chosen. The next line of new code establish this position, **pos.**

$$\textbf{pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10;}$$

This is an example of the **WTp3** structure type. The **X, Y** and **Z** are defined constants and will become **1, 2,** and **3,** but written as is helps to show the world axis coordinate relationship. Similarly, the orientation **ornt** is defined by the four components of the quaternion.

$$\textbf{ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;}$$

This quaternion corresponds to alignment with the world reference frame.

All the variables needed are now defined. The function **WTobject_newblock()** is used to create the new graphical object. The parameters required to be provided for the function call are three separate lengths, one for each of the local axis and two flags represented by defined constants for backface rejection and fastmerge. Fastmerge is a graphics parameter that deals with rendering speeds and detail. Required parameters for all WTK function calls are given in the user manual.

$$\textbf{block = WTobject\_newblock(1, 2, 3, FALSE, FALSE);}$$

A new block is created and the **WTobject** type defined structure is pointed to by **block.** This produces a 1 by 2 by 3 unit block with backfaces rejected and fastmerge negative. The position of the block is set with the **WTobject_setposition()** function.

$$\textbf{WTobject\_setposition(block, pos);}$$

24

The required parameters for this function are a pointer to the graphical object structure and the **WTp3** position. Setting the orientation is completed similarly, except the second parameter is a **WTq** orientation.

WTobject_setorientation(block,ornt);

Position and orientation can be directly loaded into a simulation with a graphical object. DXF and NFF file formats include the position and orientation of the graphical object when it was saved into that respective format. Loading a graphical object is slightly different than the example in Figure 3.3. First and foremost a file containing the graphical object must be created or made available. CAD programs are of great assistance in this endeavor. The application will not run properly if a file is not available.

D. ADDING SENSORS:

The simulation to this point is still just a static display. Sensors can allow input and interaction with the simulation. In this section, the spaceball will be coupled or "attached" to the viewpoint to permit the simulation to be viewed from different vantage points. The position and orientation of the viewpoint will then be controlled by the spaceball. Another use of a sensor is to use it to chose an object. This section will also use the mouse to chose an object and "attach" it to the spaceball. The spaceball will then control the object's position and orientation.

Figure 3.4 adds the necessary three lines of code to the program in Figure 3.3 to attach the viewpoint to the spaceball. First a new pointer, **spaceball**, to a **WTsensor** type structure is declared in the header section.

WTsensor *spaceball;

The pointer **spaceball** is set equivalent to the pointer **sensor**.

```
                    /* SENSOR ATTACHED TO VIEWPOINT SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

        static WTsensor *sensor;

/* Global declaration of a pointer to a WTsensor type structure. */
        static WTsensor *spaceball;

        static void actions();
        static WTobject *block;

/* MAIN PROGRAM */

    main()
    {
        WTp3 pos;
        WTq ornt;
                    pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;
                    ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;

                    WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

        block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
        WTobject_setposition(block,pos);
        WTobject_setorientation(block,ornt);

                    sensor = WTspaceball_new(COM1);

/* Assigns the structure pointed to by "spaceball" equal to the structure pointed to by "sensor". */
                    spaceball = sensor;

/* Establishes the proportionality of movement between the spaceball and the simulation. */

        WTsensor_setsensitivity(sensor, 0.1 * WTuniverse_getradius());

/* Attaches the spaceball to the viewpoint. */
        WTviewpoint_addsensor(WTuniverse_getviewpoint(), spaceball);

                    WTuniverse_ready();
                    WTuniverse_setactions(actions);
                    WTuniverse_go();

                    WTuniverse_delete();
    }

        /* UNIVERSE ACTIONS */

        static void actions()
        {

                    if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                    {
                            printf("BUTTON 1 ACCEPTED \n");
                            WTuniverse_stop();
                    }

        }
```

Figure 3.4

spaceball = sensor;

Next, the proportionality of movement of the input device to movement in the simulation is established.

WTsensor_setsensitivity(sensor, 0.1 *WTuniverse_getradius());

The statement above sets the proportion of the pointer sensor to one tenth the radius of the simulation universe. Finally, the sensor is attached to the graphical object with the last new line of code.

WTviewpoint_addsensor(WTuniverse_getviewpoint(), spaceball);

The function call above is actually a function call within a function call. **WTuniverse_getviewpoint()** is a WTK function call that returns the required information structure for the view before calling the **WTviewpoint_addsensor()** function.

The second sensor program is contained in Figure 3.5. A pointer to a second **WTsensor** type structure is required because the mouse is used in the simulation along with the spaceball.

static WTsensor *mouse;

Also, a second object is added to demonstrate the contrast between a graphical object attached to a sensor and one that is not. The second graphical object is created using the same techniques used to create the first. The changes have been commented in the program, but will not be individually addressed in the text here. The next alteration to the program not associated with the adding of a second graphical object is the mouse function call. This allows the input device, the mouse, to provide data to the structure pointer **mouse**.

mouse = WTmouse_new();

The next two new statements are very similar to statements already introduced.

if(WTsensor_getmiscdata(mouse)&WTMOUSE_LEFTBUTTON)

27

```
                    /* MOUSE AND SPACEBALL SENSOR SAMPLE */
/* HEADER SECTION */
#include<stdio.h>
#include"wt.h"
#include"wt.p"
            static WTsensor *sensor;
/* Global declaration of a pointer to a WTsensor type structure "mouse". */
            static WTsensor *mouse;
            static void actions();
            static WTobject *block;
/* Global declaration of a pointers to a WTobject type structure "sphere" and "curr". */
        static WTobject *sphere;
        static WTobject *curr;
/* MAIN PROGRAM */
        main()
        {
/* Local declaration of the variables "pos" and "pos1" WTp3 type structures. */
            WTp3 pos , pos1;
/* Local declaration of the variable "ornt" and "ornt1" WTq type structures. */
            WTq ornt, ornt1;
                        pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;
/* Establishes "pos1" = (10,0,10) */
                        pos1[X] = 5.0; pos1[Y] = 0.0; pos1[Z] = 10.0;
                        ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;
/* Establishes "ornt1" = (0,0,0,1) */
                        ornt1[X] = 0.0; ornt1[Y] = 0.0; ornt1[Z] = 0.0; ornt1[W] = 1.0;
                        WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);
            block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
/* Creates a graphical object and assigns "sphere" to point to it's location in memory. */
            sphere = WTobject_newsphere(1, 2, 4, FALSE, FALSE, FALSE);
            WTobject_setposition(block,pos);
/* Positions the graphical object pointed to by "sphere" at (5,0,10) */
            WTobject_setposition(sphere,pos1);
            WTobject_setorientation(block,ornt);
/* Orients the graphical object pointed to by "sphere" with the quaternion "ornt". */
            WTobject_setorientation(sphere,ornt);
                        sensor = WTspaceball_new(COM1);
/* Establishs the mouse for use. */
            mouse=WTmouse_new();
                        WTuniverse_ready();
                        WTuniverse_setactions(actions);
                        WTuniverse_go();
                        WTuniverse_delete();
        }
            /* UNIVERSE ACTIONS */
            static void actions()
            {
                    if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                        {
                printf("BUTTON 1 ACCEPTED \n");
                        WTuniverse_stop();
                        }
/* Pick object subroutine using a mouse. */
            if(WTsensor_getmiscdata(mouse)&WTMOUSE_LEFTBUTTON)
                {
/* Prints "CURRENT OBJECT SELECTED" to the screen using printf function from file stdio.h. */
                printf("CURRENT OBJECT SELECTED \n");
/* Assigns a pointer to the object closest to the mouse pointer when left button is clicked. */
                curr = WTuniverse_pickobject(*(WTp2*)WTsensor_getrawdata(mouse));
/* Sets the proportionality of movement between spaceball and the simulation. */
                        WTsensor_setsensitivity(sensor, 0.1 *WTuniverse_getradius());
/* Adds spaceball control to the object selected by curr */
                WTobject_addsensor(curr, sensor, WTFRAME_WORLD);
                        }
            }
```

Figure 3.5

28

This logical statement checks for the left mouse button much as button 1 of the spaceball was poled, and the **printf()** is called just as previously explained. The pickobject function is subsequently called.

curr = WTuniverse_pickobject(*(WTp2*)WTsensor_getrawdata(mouse));

This function returns the closest graphical object to the position of the mouse on the screen and assigns it to **curr.** After picking the object just as in the program in Figure 3.3 the sensor sensitivity is set.

WTsensor_setsensitivity(sensor, 0.1 * WTuniverse_getradius());

Finally the last statement attaches the sensor to the chosen graphical object.

WTobject_addsensor(curr, sensor, WTFRAME_WORLD);

The world axis are chosen for the reference utilizing the third parameter. When running this simulation keep in mind that there has been no provision for detaching the sensor from the graphical objects. Choosing one and then the other will allow both to be attached simultaneously.

E. ADDING LIGHTS:

The two types of lighting, ambient and directed can be demonstrated in a single example. Figure 3.6 is a modification of Figure 3.3, the core program plus a graphical object. Only slight modification is necessary to produce the results desired. First, a pointer to a **WTlight** type structure **spotlight** is declared globally.

static WTlight *spotlight;

The directed light requires a position and direction vector to be properly defined. **Pos1** and **dir** are **WTp3** variables declared locally and defined as in the previous figures. However, to illustrate one of the math conversion functions the orientation vector is

```
                        /* LIGHTS SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

        static WTsensor *sensor=NULL;
        static void actions();
        static WTobject *block;

/*Global declaration of a WTlight type structure "spotlight". */
        static WTlight *spotlight;

/* MAIN PROGRAM */

    main()
    {
/* Local declaration of WTp3 type structures "pos", "pos1" and "dir". */
        WTp3 pos, pos1, dir;
        WTq ornt;

                pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;
/* Assigns "pos1" the position (5, 0, 10). */
                pos1[X] = 5.0; pos1[Y] = 0.0; pos1[Z] = 10.0;
/* Assigns "dir" the vector (-1, 0, 0). */
                dir[X] = -1.0; dir[Y] = 0.0; dir[Z] = 0.0;

/* Twists the object 45 degrees about the Y axis and assigns the proper quaternion to "ornt". */
        WTeuler_2q(0,.785,0,ornt);

            WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);
        block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
        WTobject_setposition(block,pos);
        WTobject_setorientation(block,ornt);

                sensor = WTspaceball_new(COM1);

/* Set ambient light to maximum to .1 */
        WTlight_setambient(.1);

/* Add a directed light to position pos1 and directed at the object with .6 intensity ". */
        spotlight = WTlight_new(pos1, dir, .6);

                WTuniverse_ready();
                WTuniverse_setactions(actions);
                WTuniverse_go();

                WTuniverse_delete();
    }

                                        /* UNIVERSE ACTIONS */

        static void actions()
        {

                if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                {
            printf("BUTTON 1 ACCEPTED \n");
                        WTuniverse_stop();
        }
        }
```

Figure 3.6

30

assigned its value using Euler angles given in radians and the Euler to quaternion conversion function.

WTeuler_2q(0, .785 , 0, ornt);

This is a 45 degree angle twist about the Y axis and will assist in showing the lighting on the block by exposing the right side from the current viewpoint. To set the ambient light to a value different from the default intensity, .4, a single line of code is required.

WTlight_setambient(.1);

The intensity can be from 0.0 to 1.0 and a value of 0.1 will accentuate directed lighting. The final new statement required is the creation of the directed light. It will be placed at **pos1** which is to the object's right and directed toward the object.

spotlight = WTlight_new(pos1, dir, .6);

The parameters are position, direction, and intensity respectively. There is no limit to the number of lights in a simulation.

F. ADDING VIEWPOINTS:

Some of the viewpoint functions have already been demonstrated in section C. In addition to coupling viewpoints with sensors or objects it is possible to place them in any position and orientation desired. Figure 3.7 contains a sample program for viewpoint placement, which is a modification of Figure 3.3. A pointer to a **WTviewpoint** type structure is required, and therefore **view** is declared globally.

static WTviewpoint *view;

A position, **pos1** and an orientation, **ornt** are declared and assigned as previously demonstrated. The pointer **view** is established as the universe viewpoint with the get viewpoint function.

view = WTuniverse_getviewpoint();

31

```
                    /* VIEWPOINT SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

          static WTsensor *sensor;
          static void actions();
          static WTobject *block;

/* Global declaration of a pointer to a WTviewpoint type structure "view". */
          static WTviewpoint *view;

/* MAIN PROGRAM */

     main()
     {
/* Local declaration of WTp3 type structures "pos" and "pos1". */
          WTp3 pos, pos1;

          WTq ornt;

                    pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;
/* Positions the viewpoint at (5, -3, -10). */
                    pos1[X] = 5.0; pos1[Y] = -3; pos1[Z] = -10.0;

                    ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;

                    WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

          block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
          WTobject_setposition(block,pos);
          WTobject_setorientation(block,ornt);

                    sensor = WTspaceball_new(COM1);

/* Establishes "view" as the universe viewpoint. */
                    view = WTuniverse_getviewpoint();

/* Positions the pointer "view" at "pos1". */
          WTviewpoint_setposition(view, pos1);

/* Aligns the pointer "view" with "ornt". */
          WTviewpoint_setorientation(view, ornt);

                    WTuniverse_ready();
                    WTuniverse_setactions(actions);
                    WTuniverse_go();

                    WTuniverse_delete();
     }

          /* UNIVERSE ACTIONS */

          static void actions()
          {
                    if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                    {
               printf("BUTTON 1 ACCEPTED \n");
                              WTuniverse_stop();
          }
          }
```

Figure 3.7

32

A new and separate viewpoint could be created using the **WTviewpoint_new()** function, but this would not effect the default window view that was defined when the universe was created and is currently the universe view. The position and orientation of **view** is established with the last two new lines in a similar manner to that of graphical objects.

WTviewpoint_setposition(view, pos1);
WTviewpoint_setorientation(view, ornt);

Although only one viewpoint per window can be used at a time many can be defined to allow the simulation to shift vantage points as required. This is a powerful and useful feature.

G. ADDING TERRAIN:

The ability to produce terrain for a simulation without great difficulty is very useful. The viewpoint sample Figure 3.7, has been modified in Figure 3.8 to illustrate the terrain feature. In previous modifications the sample programs have simply added lines of code to the existing program. In this example the lines for the graphical object pointer declaration and for the graphical object creation are replaced to produce the new sample. The pointer **flat**, which is a **WTobject** type structure is declared globally.

static WTobject *flat;

It is interesting to note that terrain uses a **WTobject** type structure even though it is considered its own class of functions. Flat terrain is created using the flat terrain function, however variable altitude terrain can be produced similarly with **WTterrain_random** and **WTterrain_load** function calls.

flat = WTterrain_flat(0.1, 40, 40, -20, 20, 1, 1, 0x000, 0xFFF, FALSE);

The parameters contained are altitude, length, width, positioning(X and Z) , polygon construction size(X and Z), colors and backface rejection information.

33

```
                /* TERRAIN SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

        static WTsensor *sensor;
        static void actions();
    static WTviewpoint *view;

/* Global declaration of a WTobject type structure "flat". */
        static WTobject *flat;

/* MAIN PROGRAM */

    main()
    {
        WTp3 pos1;
        WTq ornt;

                pos1[X] = -5.0; pos1[Y] = -3.0; pos1[Z] = -10.0;
                ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;

                WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

/* Creates a flat terrain with specifications and positioning contained in the parameters chosen. */
        flat = WTterrain_flat(0.1,40,40,-20,20,1,1,0x000,0xFFF,FALSE);

                sensor = WTspaceball_new(COM1);

                view = WTuniverse_getviewpoint();
        WTviewpoint_setposition(view,pos1);
        WTviewpoint_setorientation(view,ornt);

                WTuniverse_ready();
                WTuniverse_setactions(actions);
                WTuniverse_go();

                WTuniverse_delete();
    }

        /* UNIVERSE ACTIONS */

        static void actions()
        {
                if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                {
            printf("BUTTON 1 ACCEPTED \n");
                        WTuniverse_stop();
        }
        }
```

Figure 3.8

34

H. ADDING PATHS:

Paths are a feature that allow viewpoints and objects to move along set

trajectories. Figure 3.9 is a sample program that is a modification of the terrain sample

Figure 3.8. The first modification to 3.8 was that the graphical object was included from

Figure 3.3 the graphical object sample. When creating a path, the path and pathnodes

must first be created and associated and then an object or viewpoint can be associated

with them. Paths are then played similarly to an audio tape, the contents are executed in

order. The first modification required is the global declaration of a **WTpath** type

structure **path**.

static WTpath *path;

Global declarations for two pointers for nodes that define the path are also needed.

static WTpathnode *node1;
static WTpathnode *node2;

Local declaration of variables is accomplished next, however a **WTpq** type structure **pq** is

declared in addition to the **WTp3** and **WTq type** structures. A **WTpq** is a combination

of position and orientation information into a single structure.

WTpq pq;

The pathnodes are created next and it should be noted that the parameter required in the

new pathnode function is a **WTpq** structure that must be referenced by address.

node1 = WTpathnode_new(&pq);
node2 = WTpathnode_new(&pq);

Now that the nodes have been created, positions and orientations are set for each **node1**

and **node2**.

WTpathnode_setposition(node1, pos);
WTpathnode_setposition(node2, pos2);

WTpathnode_setorientation(node1, ornt);
WTpathnode_setorientation(node2, ornt1);

35

```
                    /* PATH SAMPLE */
/* HEADER SECTION */
#include<stdio.h>
#include"wt.h"
#include"wt.p"
            static WTsensor *sensor=NULL;
            static void actions();
            static WTviewpoint *view;
            static WTobject *flat;
            static WTobject *block;
/* Global declaration of WTpath type structure "path". */
            static WTpath *path;
/* Global declaration of WTpathnode type structures "node1" and "node2". */
            static WTpathnode *node1;
            static WTpathnode *node2;


/* MAIN PROGRAM */
    main()
    {
/*Local declaration of WTp3 and WTq type variables. */
        WTp3 pos, pos1, pos2;
        WTq ornt, ornt1;


/*Local declaration of a WTpq type structure "pq". */
        WTpq pq;
/* Assigns "pos" at (0,0,30), "pos1" at (-5,-3, 10) and "pos2" at (10,0,20). */
        pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 30.0;
         pos1[X] = -5.0; pos1[Y] = -3.0; pos1[Z] = 10.0;
         pos2[X] = 10.0; pos2[Y] = 0.0; pos2[Z] = 50.0;
/* Aligns "ornt" with world axis and rotates "ornt1" 90 degrees about the X axis. */
         ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;
         ornt1[X] = 1.0; ornt1[Y] = 0.0; ornt1[Z] = 0.0; ornt1[W] = 0.0;
         WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);
         flat = WTterrain_flat(0.1, 40, 40, -20, 20, 1, 1, 0x000, 0xFFF, FALSE);
         block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
         WTobject_setposition(block,pos);
         WTobject_setorientation(block,ornt);
         sensor = WTspaceball_new(COM1);
         view = WTuniverse_getviewpoint();
         WTviewpoint_setposition(view,pos1);
         WTviewpoint_setorientation(view,ornt);
/* Establish pathnodes "node1" and "node2". (Must assign by address. The "&" is mandatory.) */
         node1 = WTpathnode_new(&pq);
         node2 = WTpathnode_new(&pq);
/* Set positions of "node1" and "node2". */
         WTpathnode_setposition(node1,pos);
         WTpathnode_setposition(node2,pos2);
/* Set orientations of "node1" and "node2". */
         WTpathnode_setorientation(node1,ornt);
         WTpathnode_setorientation(node2,ornt1);
/* Establish path "path". */
         path = WTpath_new(block);
/* Add nodes to path. */
         WTpath_appendnode(path,node1);
         WTpath_appendnode(path,node2);
/* Smooth the path with more nodes between "node1" and "node2". */
         path = WTpath_interpolate(path, 20, WTPATH_LINEAR);
         WTuniverse_ready();
         WTuniverse_setactions(actions);
         WTuniverse_go();
         WTuniverse_delete();
    }
         /* UNIVERSE ACTIONS */
         static void actions()
         {
                    if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                    {
                    printf("BUTTON 1 ACCEPTED \n");
                    WTuniverse_stop();
                    }

                    if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON2)
                    {
                    printf("BUTTON 2 ACCEPTED \n");
/* Ensure path starts at the first node in the path. */
         WTpath_rewind(path);
/* Attach "block" to the path. */
         WTpath_setobject(path, block);
/* Start along the path. */
         WTpath_play(path);
                    }
                    }
```

Figure 3.9

In this example, the orientation of the nodes are different as well as the positions. A path is then created utilizing the new path function.

path = WTpath_new(block);

The parameter in the function must be a graphical object and is used to mark the pathnodes when the path visualization function, not used in this example, is called. At this point all the nodes and the path are defined, but they are not associated. The path has no pathnodes associated with it. To add **node1** and **node2**, in order, to **path** the append node functions are called.

WTpath_appendnode(path, node1);
WTpath_appendnode(path, node2);

The path is now complete although almost trivial with only two nodes. More nodes can be interpolated between the points of a path using the path interpolate function.

path = WTpath_interpolate(path, 20, WTPATH_LINEAR);

The parameters are the path to be modified, in this case the same path, the number of points to be included, and the method of interpolation. This statement produces a smoothed path between **node1** and **node2**. For the example given, now all the defined parameters are complete the path can be played. The **actions()** function and specifically Button 2 of the spaceball has been chosen to play the path, however this is not required and it could have been initiated in the **main()** function or any other. To ensure that the path is started at the first node the rewind function is called.

WTpath_rewind(path);

Next a graphical object is associated or attached to the path.

WTpath_setobject(path, block);

Finally, the object is set in motion along the path with the play function.

WTpath_play(path);

37

The example given is a rudimentary path example and much more complex and intricate applications are possible using the basic concepts presented.

I. ADDING WINDOWS:

In all of the samples given there has only been one view presented at a time. Multiple views can be displayed by opening additional windows. Figure 3.10 provides a sample program that has been modified from Figure 3.7. The modified program provides two windows and two views of the graphical object **block** simultaneously. Initially a pointer to a **WTwindow** type structure **window1** is globally defined.

static WTwindow *window1;

Next a new viewpoint is defined using the viewpoint new function vice using the universe view. At this point there are two separate views available, the universe view and the new view.

view = WTviewpoint_new();

As before the viewpoint position and orientation are assigned. A new window, **window1** is established. This does not replace the default window, but is created in addition to it.

window1 = WTwindow_new(500, 500, 500, 500, FALSE);

The parameters are length, width, upper left X and Z screen coordinates all in pixels, and a platform specific flag. This window is 500 by 500 pixels with the upper left corner at position (500, 500) pixels on the screen. Finally, **window1** is assigned the viewpoint **view**, which is different, moved to the right and up, from the default view.

WTwindow_setviewpoint(window1, view);

```
/* WINDOW SAMPLE */

/* HEADER SECTION */

#include<stdio.h>
#include"wt.h"
#include"wt.p"

        static WTsensor *sensor;
        static void actions();
        static WTobject *block;
        static WTviewpoint *view;

/* Global declaration of a WTwindow type structure "window1". */
        static WTwindow *window1;

/* MAIN PROGRAM */

    main()
    {
        WTp3 pos, pos1;
        WTq ornt;

                pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 10.0;
                pos1[X] = 5.0; pos1[Y] = -3; pos1[Z] = -10.0;

                ornt[X] = 0.0; ornt[Y] = 0.0; ornt[Z] = 0.0; ornt[W] = 1.0;

                WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);

        block = WTobject_newblock(1, 2, 3, FALSE, FALSE);
        WTobject_setposition(block,pos);
        WTobject_setorientation(block,ornt);

                sensor = WTspaceball_new(COM1);

/* Establishes a new view seperate from the universe view. */
                view = WTviewpoint_new();

        WTviewpoint_setposition(view, pos1);
        WTviewpoint_setorientation(view, ornt);

/* Establish a new window "window1". */
                window1 = WTwindow_new(500,500,500,500, FALSE);

/* Set "view" as the viewpoint in "window1". */
                WTwindow_setviewpoint(window1,view);

                WTuniverse_ready();
                WTuniverse_setactions(actions);
                WTuniverse_go();

                WTuniverse_delete();
    }
        /* UNIVERSE ACTIONS */

        static void actions()
        {
                if (WTsensor_getmiscdata(sensor)&WTSPACEBALL_BUTTON1)
                {
           printf("BUTTON 1 ACCEPTED \n");
                        WTuniverse_stop();
        }
        }
```

Figure 3.10

J. USER DEFINED FUNCTIONS:

It would be impossible to predetermine every function that could be needed to produce every application, therefore WTK allows the user to define functions to accomplish functions not already defined. In the standard form these are placed after the **main**() function and preferably also after the **actions**() function. Care must be taken by the user to ensure that variables passed between functions or changed within a function are handled properly to achieve the desired result. Examples of user defined functions are included in the application provided in Appendix A.

IV. EVALUATION SCENARIO

A. GOALS:

As stated in Chapter II, the broad objective of the scenario is to demonstrate that virtual reality software can assist in the ability to make decisions. A military scenario was chosen for this application to demonstrate a situation of immediate and substantial consequence. The Combat Information Center (CIC) scenario written for this application and provided as Appendix A, is a basic example of some of the capabilities provided by virtual reality software. The intent was to demonstrate the software as a valuable addition to current systems such as the Naval Tactical Data System (NTDS) and to show a few of WTK's capabilities. This software is not a replacement for the current systems, but a means of collecting and displaying the information already available. The application created involved two forces with multiple assets, formed into battle groups. One battle group was set travelling Northeast the other stood in its path. Although the scenario contained actual combatants of the United States and the former Soviet Union, all information was kept unclassified. Also to ensure that no classified strategies or doctrine were exposed, weapon deployments were not included.

The CIC is where data collected is displayed and tactical decision makers gather to formulate and execute battle strategies. In this space there are numerous status boards, radar displays, electronic emissions displays, acoustical data collections, and more. Each piece of information is important. Doctrines are used for structuring decisions and to provide the decision maker with guidelines. If a complete and total picture of events can be presented to the decision maker, the number of decisions that are made in conflict with these guidelines will be reduced. The problem is that in many instances only a portion of

the information is available or acknowledged by the decision maker even though it has been collected and perhaps even processed and displayed.

B. FEATURES:

There are several WTK features incorporated. The application is produce on a video monitor divided into four sections. An example of the video display format is provided in Figure 4.1. The upper left display in this figure is the "overview window" providing a pan view of the scenario, similar to a 2D display. The lower left display is the "universe window" with a view that can be positioned at any location and in any orientation within the scenario. The upper right or "object window" can either be the same as the "universe window" or attached to a graphical object to provide a view as seen from that graphical object's position and orientation. Finally, the last section, bottom right in Figure 4.1, contains the "command and data window." The "command and data window" provides a means for text input and output. This format of display was chosen for the application to best demonstrate the features programmed. The display is extremely versatile and should be driven by the application requirements.

The 3D display included a terrain grid representing a large swath of ocean. This facilitates immediate recognition of relative distances visually. The many combatants included in the scenario, were assumed to be positionally updated by automatic sensor update or through the latest intelligence data manually. These units were given shapes similar to the 2D symbols used on NTDS displays to provide some continuity with current systems. However, specific symbols representing ship type, weapons capabilities or any number of other immediately recognizable and useful information could be incorporated. The color scheme red and blue is also commonly used in military scenarios, yet not available on NTDS systems.
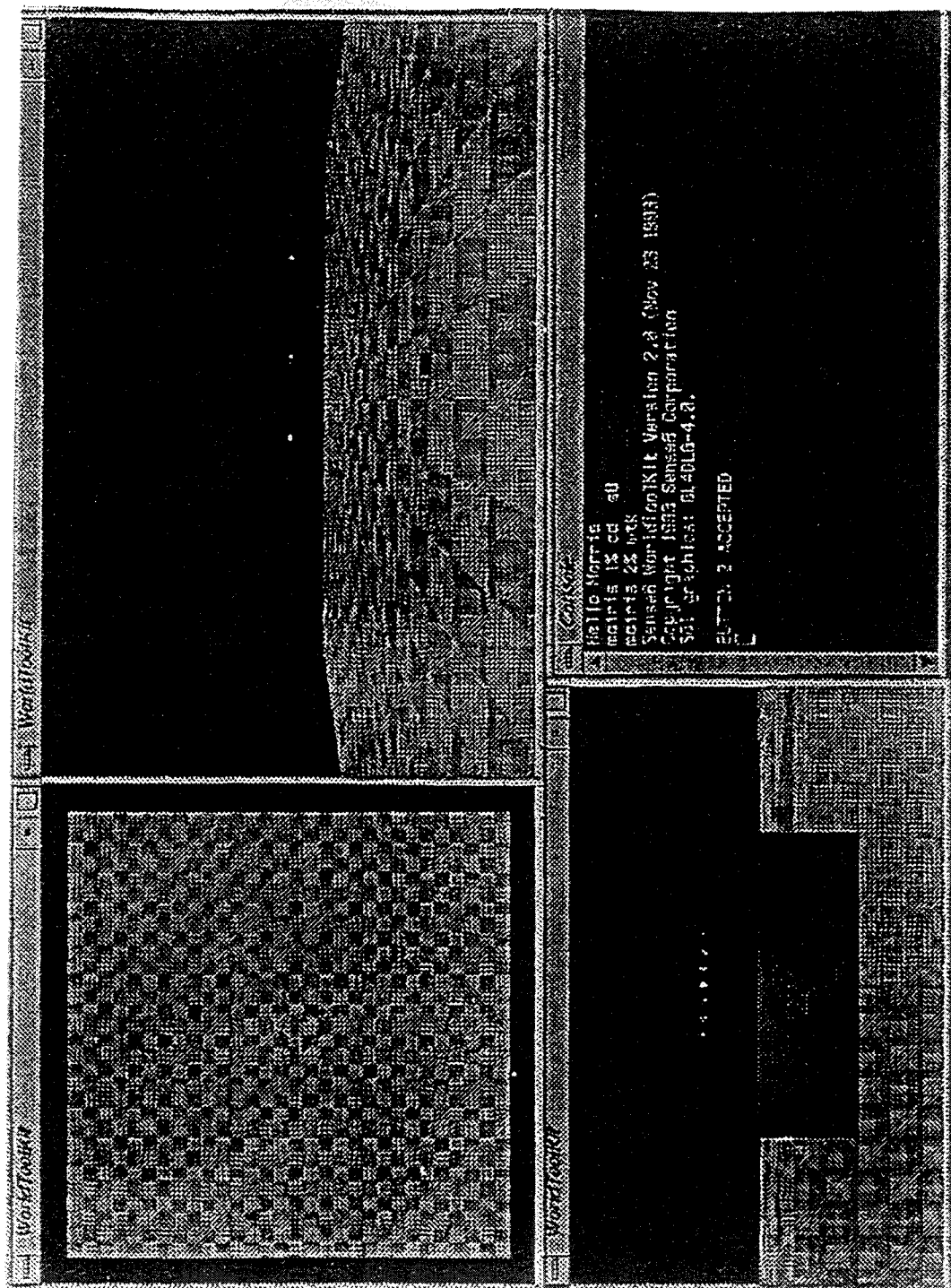
Figure 4.1

43

Information displayed on the video monitor can become cluttered and unreadable due to high concentrations of displayed data. For this reason some features should be available only on demand. Sensors provide a means of accessing specifically programmed features through the **actions**() function. Figure 4.2 is a chart of user inputs through a sensor to the specific programmed responses for the application provided as Appendix A.

SENSOR	BUTTON	ACTION
spaceball	1	ends application
spaceball	2	begins to play defined paths
spaceball	3	selects object closest to mouse arrow displays data associated with selected object
spaceball	4	assigns universe view to "object window"
spaceball	6	selects object closest to mouse arrow permits selected objects path to be altered
spaceball	7	attaches the universe view to the spaceball
mouse	left	from the "pan view" while altering a path, selects a 2D point to be used as a new node

Figure 4.2

Button 1 on the spaceball accesses the WTuniverse_stop() function call ends the application, while Button 2 plays all defined paths and attaches the designated objects to the paths. Data printouts on a video monitor are currently available on NTDS systems and are also included in the scenario. The actions required to display the information is similar on both systems. On NTDS a cursor is positioned over the video return on a radar 2D display and then by depressing the proper button the information associated with the video return is displayed. In this application the mouse is used to position the pointer over the graphical object on the screen and depressing spaceball Button 3 will display the associated information, course, speed, and platform. The information is displayed in the

"command and data window." Information is not limited to these three categories, which are provided as a demonstration of the feature.

Selecting a graphical object as described using the mouse and spaceball Button 3 or Button 6 will attach the "object window" viewpoint to graphical object and provide a view from the graphical object's position and orientation. Button 4 on the spaceball makes the "object window" viewpoint the same as the "universe window." In addition to changing the "object window" viewpoint, Button 6 also allows the current path that an object is on to be changed. If this option is chosen, the "command and data window" displays a series of questions that lead the user though the steps required to change the path as desired. This feature is to show the ability of WTK to be used as a training or strategy generation medium. A scenario can be run several times, each time changing the scenario while running, revealing the effect of different unit deployments or actions.

Two dimensional displays have limited display capabilities. This application showcases the ability to look at the scenario from any vantage point. By enabling the viewpoint to move by depressing spaceball Button 7, the spaceball can be used to position the viewpoint of the "universe window" to any position or orientation desired.

A flow chart showing the general flow of the application in Appendix A is provided as Figure 4.3.
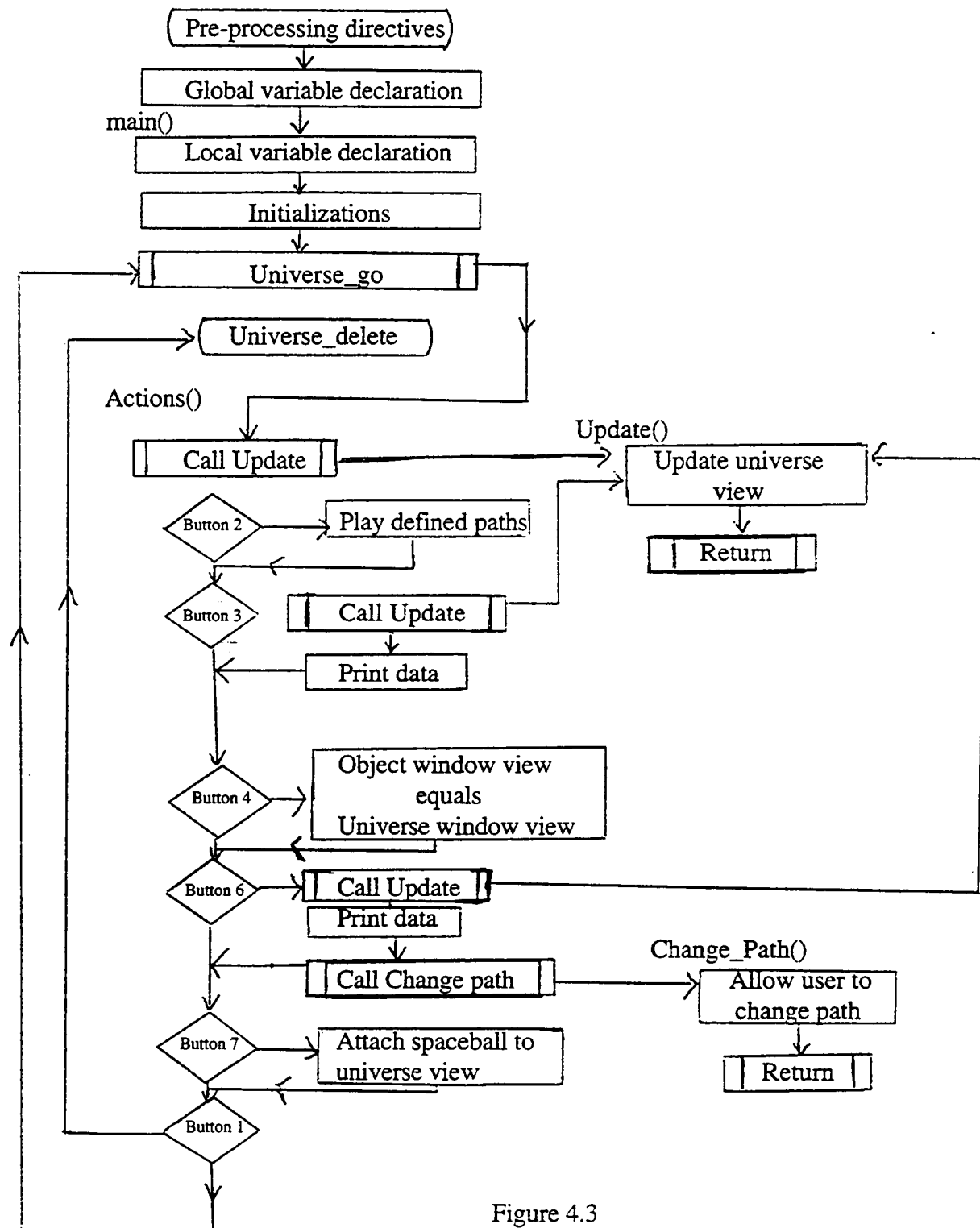
CIC SCENARIO FLOWCHART

Figure 4.3

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS:

WTK is just one of many available off the shelf software packages that can provide enhanced data and information display. The features and advantages presented in this thesis provide a glimpse at the possibilities available.

-Three dimensional representation provides a display that is familiar and more easily recognizable for the decision making process.

-Using off the shelf software more information than currently available on NTDS and other systems can be displayed in an uncluttered information dense format.

-Information can be displayed in real time using WTK or other similar software making the concept of using virtual reality software in the decision making environment tactically significant.

B. RECOMMENDATIONS:

WTK and other virtual reality softwares require some modification or programming to be adapted for use. These softwares are tools to produce a versatile and improved informational display. Tools require training for proper utilization, and virtual reality software is no different. Any tool that is too difficult to use is generally not used in common practice, and therefore it is recommended that user friendly subroutines and functions be written into each scenario. An example of this technique is the subroutine for changing the path of a graphical object in the application provided in Appendix A. The subroutine walks the user through each step of the process with a series of simple questions and checks the correctness of each input.

It is further recommended that libraries of subroutines and functions germane to the types of applications being constructed be made available to the user. Then building an application could be as simple as including the proper previously written subroutines. This would still allow the user to write a custom function or subroutine if a pre-existing function for exactly what was trying to be accomplished was not among the functions provided.

Synthetic environment software is relatively new and there is much that was not addressed or only briefly mentioned. Further research areas include motion sensitive input devices, attaching viewpoints to head or hand movement, determining optimal presentation density and display format, study to determine if and by how much decision making is improved in actual situations, and what is the best software available with regard to cost and performance.

APPENDIX A

/*CIC SCENARIO*/

/* PRE-PROCESSING DIRECTIVES*/
```
#include<stdio.h>
#include"wt.h"
#include"wt.p"
#include<math.h>
#define max 25
```

/* GLOBAL VARIABLE DECLARATION*/
```
        static void actions();

    static float oneknot;

    static WTpathnode *tempnode;
    static WTpath *tempname;

        static WTsensor *spaceball=NULL;
        static WTsensor *sensor=NULL;
    static WTsensor *mouse=NULL;

    static WTobject *nodeobj;
    static WTobject *curr;

    static WTobject *bb;
    static WTobject *cg1;
    static WTobject *cg2;
    static WTobject *cg3;
    static WTobject *dd1;
    static WTobject *dd2;
    static WTobject *dd3;
    static WTobject *ff;
    static WTobject *oa;
    static WTobject *h1;
    static WTobject *h2;
    static WTobject *h3;
    static WTobject *ftr1;
    static WTobject *ftr2;
    static WTobject *ftr3;
```

```c
static WTobject *ftr4;
static WTobject *ftr5;
static WTobject *ftr6;
static WTobject *ftr7;
static WTobject *ftr8;
static WTobject *tnkr;

static WTobject *bm1;
static WTobject *bm2;
static WTobject *bm3;
static WTobject *bm4;
static WTobject *bm5;
static WTobject *bm6;
static WTobject *bm7;
static WTobject *bm8;
static WTobject *bm9;
static WTobject *bm10;
static WTobject *bm11;
static WTobject *bm12;
static WTobject *bm13;
static WTobject *bm14;
static WTobject *bm15;
static WTobject *bm16;
static WTobject *bm17;
static WTobject *bm18;
static WTobject *bm19;
static WTobject *bm20;
static WTobject *bm21;
static WTobject *bm22;
static WTobject *bm23;
static WTobject *bm24;
static WTobject *kirv;
static WTobject *slv;
static WTobject *sov1;
static WTobject *sov2;
static WTobject *kres1;
static WTobject *kres2;
static WTobject *ol;

static WTpath *slide;
static WTpath *bomer1;
static WTpath *bomer2;
static WTpath *fight1;
```

```c
        static WTpath *fight3;
        static WTpath *fight5;
        static WTpath *fight7;
        static WTpath *tanker1;
        static WTpath *helo1;
        static WTpath *helo2;
        static WTpath *helo3;

        static WTpathnode *nextnode;
        static WTpathnode *currnode;

        static WTviewpoint *view;
        static WTviewpoint *obview;
        static WTviewpoint *overview;

        static WTwindow *window1;
        static WTwindow *window2;

        static WTq overornt;

        static struct data
           {
           WTpath *currpath;
        int index;
           int speed;
        int course;
        int speedkts;
        char *name;
           } Data, DATA[50];

        static WTmouse_rawdata *raw;


/*MAIN*/
 main()
{
/*LOCAL VARIABLE DECLARATION*/

        WTpq pq;

        WTp3 pbb,pcg1,pcg2,pcg3,pdd1,pdd2,pdd3;
        WTp3 pff,poa,ph1,ph2,ph3,overpos,overdir;
```

51

```
WTp3 pbm1,pbm2,pkirv,pslv,psov1,psov2;
WTp3 pkres1,pkres2,pol;

WTp3 pbm3,pbm4,pbm5,pbm6,pbm7,pbm8,pbm9,pbm10,pbm11;
WTp3 pbm12,pbm13,pbm14,pbm15,pbm16,pbm17,pbm18,pbm19;
WTp3 pbm20,pbm21,pbm22,pbm23,pbm24;

WTp3 pftr1,pftr2,pftr3,pftr4,pftr5,pftr6,pftr7,pftr8,ptnkr;

WTq rotatex;
```

/*INITIALIZATIONS*/

```
overpos[X]= 0;overpos[Y]= -200;overpos[Z]= 0;
overdir[X]= 0;overdir[Y]= 1;overdir[Z]= 0;
overornt[X]= 0;overornt[Y]= 0;overornt[Z]= 0;overornt[W]= 1;
pbb[X]= -10;pbb[Y]= 0.5;pbb[Z]= -10;
pcg1[X]= -10;pcg1[Y]= 0.5;pcg1[Z]= -5.0;
pcg2[X]= -5;pcg2[Y]= 0.5;pcg2[Z]= 60.0;
pcg3[X]= 65;pcg3[Y]= 0.5;pcg3[Z]= -5.0;
pdd1[X]= -15;pdd1[Y]= 0.5;pdd1[Z]= -22;
pdd2[X]= 0;pdd2[Y]= 0.5;pdd2[Z]= 5.0;
pdd3[X]= 10;pdd3[Y]= 0.5;pdd3[Z]= -10.0;
pff[X]= 15;pff[Y]= 0.5;pff[Z]= -22.0;
poa[X]= 0;poa[Y]= 0.5;poa[Z]= -20.0;
pftr1[X]= -50;pftr1[Y]= -20.0;pftr1[Z]= -20;
pftr2[X]= -48;pftr2[Y]= -20.0;pftr2[Z]= -18;
pftr3[X]= -90;pftr3[Y]= -20.0;pftr3[Z]= -90;
pftr4[X]= -87;pftr4[Y]= -20.0;pftr4[Z]= -91;
pftr5[X]= 20;pftr5[Y]= -20.0;pftr5[Z]= -120;
pftr6[X]= 23;pftr6[Y]= -20.0;pftr6[Z]= -123;
pftr7[X]= -20;pftr7[Y]= -20.0;pftr7[Z]= -140;
pftr8[X]= -17;pftr8[Y]= -20.0;pftr8[Z]= -142;
ptnkr[X]= -147;ptnkr[Y]= -20.0;ptnkr[Z]= -146;
ph1[X]= -17;ph1[Y]= -5;ph1[Z]= -35;
ph2[X]= -20;ph2[Y]= -5;ph2[Z]= 18.0;
ph3[X]= 20;ph3[Y]= -5;ph3[Z]= -16.0;
pbm1[X]= 0;pbm1[Y]= -20.0;pbm1[Z]= 140;
pbm3[X]= 6;pbm3[Y]= -20.0;pbm3[Z]= 140;
pbm4[X]= -6;pbm4[Y]= -20.0;pbm4[Z]= 140;
pbm5[X]= -12;pbm5[Y]= -20.0;pbm5[Z]= 140;
pbm6[X]= -18;pbm6[Y]= -20.0;pbm6[Z]= 140;
pbm7[X]= 12;pbm7[Y]= -20.0;pbm7[Z]= 140;
```

```
pbm8[X]= 18;pbm8[Y]= -20.0;pbm8[Z]= 140;
pbm9[X]= 0;pbm9[Y]= -20.0;pbm9[Z]= 146;
pbm10[X]= 6;pbm10[Y]= -20.0;pbm10[Z]= 146;
pbm11[X]= 12;pbm11[Y]= -20.0;pbm11[Z]= 146;
pbm12[X]= -6;pbm12[Y]= -20.0;pbm12[Z]= 146;
pbm13[X]= -12;pbm13[Y]= -20.0;pbm13[Z]= 146;
pbm2[X]= 135;pbm2[Y]= -20;pbm2[Z]= 20.0;
pbm14[X]= 131;pbm14[Y]= -20;pbm14[Z]= 26.0;
pbm15[X]= 139;pbm15[Y]= -20;pbm15[Z]= 14.0;
pbm16[X]= 141;pbm16[Y]= -20;pbm16[Z]= 24.0;
pbm17[X]= 135;pbm17[Y]= -20;pbm17[Z]= 28.0;
pbm18[X]= 141;pbm18[Y]= -20;pbm18[Z]= 16;
pbm19[X]= 143;pbm19[Y]= -20;pbm19[Z]= 24.0;
pbm20[X]= 137;pbm20[Y]= -20;pbm20[Z]= 30.0;
pbm21[X]= 143;pbm21[Y]= -20;pbm21[Z]= 18.0;
pbm22[X]= 147;pbm22[Y]= -20;pbm22[Z]= 26.0;
pbm23[X]= 139;pbm23[Y]= -20;pbm23[Z]= 32;
pbm24[X]= 145;pbm24[Y]= -20;pbm24[Z]= 20;
pkirv[X]= 110;pkirv[Y]= 0.5;pkirv[Z]= 70;
pslv[X]= 95;pslv[Y]= 0.5;pslv[Z]= 65;
psov1[X]= 100;psov1[Y]= 0.5;psov1[Z]= 80.0;
psov2[X]= 80;psov2[Y]= 0.5;psov2[Z]= 100.0;
pkres1[X]= 50;pkres1[Y]= 0.5;pkres1[Z]= 70.0;
pkres2[X]= 120;pkres2[Y]= 0.5;pkres2[Z]= 90.0;
pol[X]= 116;pol[Y]= 0.5;pol[Z]= 68;

oneknot = 1.6/3300;

WTuniverse_new(WTDISPLAY_DEFAULT,WTWINDOW_DEFAULT);


WTuniverse_setbgcolor(0x000);
WTuniverse_load("oceangrid",&pq,1);

nodeobj=WTobject_newblock(1,1,1,FALSE,FALSE);
WTobject_setvisibility(nodeobj, FALSE);

bb = WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(bb,0x00f);
cg1= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(cg1,0x00f);
cg2= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(cg2,0x00f);
```

```
cg3= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(cg3,0x00f);
dd1= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(dd1,0x00f);
dd2= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(dd2,0x00f);
dd3= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(dd3,0x00f);
ff = WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(ff,0x00f);
oa = WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(oa,0x00f);
h1 = WTobject_newsphere(.5,8,16,FALSE,FALSE,TRUE);
WTobject_setcolor(h1,0x00f);
h2 = WTobject_newsphere(.5,8,16,FALSE,FALSE,TRUE);
WTobject_setcolor(h2,0x00f);
h3 = WTobject_newsphere(.5,8,16,FALSE,FALSE,TRUE);
WTobject_setcolor(h3,0x00f);
ftr1 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr1,0xfff);
ftr2 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr2,0x00f);
ftr3 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr3,0xfff);
ftr4 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr4,0x00f);
ftr5 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr5,0xfff);
ftr6 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr6,0x00f);
ftr7 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr7,0xfff);
ftr8 = WTobject_newsphere(1,4,8,FALSE,FALSE,TRUE);
WTobject_setcolor(ftr8,0x00f);
tnkr = WTobject_newsphere(1,8,16,FALSE,FALSE,TRUE);
WTobject_setcolor(tnkr,0x00f);

bm1 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm1,0xff0);
bm2 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm2,0xff0);
bm3 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm3,0xf00);
```

```
bm4 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm4,0xf00);
bm5 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm5,0xf00);
bm6 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm6,0xf00);
bm7 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm7,0xf00);
bm8 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm8,0xf00);
bm9 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm9,0xf00);
bm10 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm10,0xf00);
bm11 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm11,0xf00);
bm12 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm12,0xf00);
bm13 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm13,0xf00);
bm14 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm14,0xf00);
bm15 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm15,0xf00);
bm16 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm16,0xf00);
bm17 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm17,0xf00);
bm18 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm18,0xf00);
bm19= WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm19,0xf00);
bm20 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm20,0xf00);
bm21 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm21,0xf00);
bm22 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm22,0xf00);
bm23 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm23,0xf00);
bm24 = WTobject_newsphere(1,2,4,FALSE,FALSE,TRUE);
WTobject_setcolor(bm24,0xf00);
kirv= WTobject_newblock(3,1,3,FALSE,FALSE);
```

```
WTobject_setcolor(kirv,0xf00);
slv = WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(slv,0xf00);
sov1= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(sov1,0xf00);
sov2= WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(sov2,0xf00);
kres1=WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(kres1,0xf00);
kres2=WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(kres2,0xf00);
ol  = WTobject_newblock(3,1,3,FALSE,FALSE);
WTobject_setcolor(ol,0xf00);

WTobject_setposition(bb,pbb);
WTobject_setposition(cg1,pcg1);
WTobject_setposition(cg2,pcg2);
WTobject_setposition(cg3,pcg3);
WTobject_setposition(dd1,pdd1);
WTobject_setposition(dd2,pdd2);
WTobject_setposition(dd3,pdd3);
WTobject_setposition(ff,pff);
WTobject_setposition(oa,poa);
WTobject_setposition(h1,ph1);
WTobject_setposition(h2,ph2);
WTobject_setposition(h3,ph3);
WTobject_setposition(ftr1,pftr1);
WTobject_setposition(ftr2,pftr2);
WTobject_setposition(ftr3,pftr3);
WTobject_setposition(ftr4,pftr4);
WTobject_setposition(ftr5,pftr5);
WTobject_setposition(ftr6,pftr6);
WTobject_setposition(ftr7,pftr7);
WTobject_setposition(ftr8,pftr8);
WTobject_setposition(tnkr,ptnkr);

WTobject_setposition(bm1,pbm1);
WTobject_setposition(bm2,pbm2);
WTobject_setposition(bm3,pbm3);
WTobject_setposition(bm4,pbm4);
WTobject_setposition(bm5,pbm5);
WTobject_setposition(bm6,pbm6);
WTobject_setposition(bm7,pbm7);
```

```
WTobject_setposition(bm8,pbm8);
WTobject_setposition(bm9,pbm9);
WTobject_setposition(bm10,pbm10);
WTobject_setposition(bm11,pbm11);
WTobject_setposition(bm12,pbm12);
WTobject_setposition(bm13,pbm13);
WTobject_setposition(bm14,pbm14);
WTobject_setposition(bm15,pbm15);
WTobject_setposition(bm16,pbm16);
WTobject_setposition(bm17,pbm17);
WTobject_setposition(bm18,pbm18);
WTobject_setposition(bm19,pbm19);
WTobject_setposition(bm20,pbm20);
WTobject_setposition(bm21,pbm21);
WTobject_setposition(bm22,pbm22);
WTobject_setposition(bm23,pbm23);
WTobject_setposition(bm24,pbm24);
WTobject_setposition(kirv,pkirv);
WTobject_setposition(slv,pslv);
WTobject_setposition(sov1,psov1);
WTobject_setposition(sov2,psov2);
WTobject_setposition(kres1,pkres1);
WTobject_setposition(kres2,pkres2);
WTobject_setposition(ol,pol);

bomer1 = WTpath_load("bomer1.pth",nodeobj);
bomer2 = WTpath_load("bomer2.pth",nodeobj);

fight1 = WTpath_load("figth1.pth",nodeobj);
fight3 = WTpath_load("fight3.pth",nodeobj);
fight5 = WTpath_load("fight5.pth",nodeobj);
fight7 = WTpath_load("fight7.pth",nodeobj);
tanker1 = WTpath_load("tanker1.pth",nodeobj);
helo1 = WTpath_load("helo1.pth",nodeobj);
helo2 = WTpath_load("helo2.pth",nodeobj);
helo3 = WTpath_load("helo3.pth",nodeobj);

DATA[1].currpath = bomer1;
DATA[1].index = 1;
DATA[1].speed = 1;
DATA[1].speedkts = 550;
DATA[1].course = 180;
DATA[1].name = "BADGER G";
```

```
WTobject_setdata(bm1,&DATA[1]);

DATA[2].currpath = bomer2;
DATA[2].index = 2;
DATA[2].speed = 1;
DATA[2].speedkts = 550;
DATA[2].course = 265;
DATA[2].name = "BACKFIRE B";

WTobject_setdata(bm2,&DATA[2]);


DATA[3].currpath = NULL;
DATA[3].index = 3;
DATA[3].speed = 1;
DATA[3].speedkts = 12;
DATA[3].course = 45;
DATA[3].name = "J F KENNEDY";

WTobject_setdata(bb,&DATA[3]);

DATA[4].currpath = NULL;
DATA[4].index = 4;
DATA[4].speed = 1;
DATA[4].speedkts = 12;
DATA[4].course = 45;
DATA[4].name = "VINCENNES";

WTobject_setdata(cg1,&DATA[4]);

DATA[5].currpath = NULL;
DATA[5].index = 5;
DATA[5].speed = 1;
DATA[5].speedkts = 12;
DATA[5].course = 45;
DATA[5].name = "CHOSIN";

WTobject_setdata(cg2,&DATA[5]);

DATA[6].currpath = NULL;
DATA[6].index = 6;
DATA[6].speed = 1;
```

```c
DATA[6].speedkts = 12;
DATA[6].course = 45;
DATA[6].name = "VALLEY FORGE";

WTobject_setdata(cg3,&DATA[6]);

DATA[7].currpath = NULL;
DATA[7].index = 7;
DATA[7].speed = 1;
DATA[7].speedkts = 12;
DATA[7].course = 45;
DATA[7].name = "LEFTWICH";

WTobject_setdata(dd1,&DATA[7]);

DATA[8].currpath = NULL;
DATA[8].index = 8;
DATA[8].speed = 1;
DATA[8].speedkts = 12;
DATA[8].course = 45;
DATA[8].name = "MERRILL";

WTobject_setdata(dd2,&DATA[8]);

DATA[9].currpath = NULL;
DATA[9].index = 9;
DATA[9].speed = 1;
DATA[9].speedkts = 12;
DATA[9].course = 45;
DATA[9].name = "O'BRIEN";

WTobject_setdata(dd3,&DATA[9]);

DATA[10].currpath = helo1;
DATA[10].index = 10;
DATA[10].speed = 1;
DATA[10].speedkts = 12;
DATA[10].course = 45;
DATA[10].name = "SH-60B";

WTobject_setdata(h1,&DATA[10]);

DATA[11].currpath = helo2;
```

```
DATA[11].index = 11;
DATA[11].speed = 1;
DATA[11].speedkts = 12;
DATA[11].course = 45;
DATA[11].name = "SH-60B";

WTobject_setdata(h2,&DATA[11]);

DATA[12].currpath = helo3;
DATA[12].index = 12;
DATA[12].speed = 1;
DATA[12].speedkts = 12;
DATA[12].course = 45;
DATA[12].name = "H-3";

WTobject_setdata(h3,&DATA[12]);

DATA[13].currpath = NULL;
DATA[13].index = 13;
DATA[13].speed = 1;
DATA[13].speedkts = 12;
DATA[13].course = 45;
DATA[13].name = "ROANOKE";

WTobject_setdata(oa,&DATA[13]);

DATA[14].currpath = NULL;
DATA[14].index = 14;
DATA[14].speed = 1;
DATA[14].speedkts = 12;
DATA[14].course = 45;
DATA[14].name = "OULETTE";

WTobject_setdata(ff,&DATA[14]);

DATA[15].currpath = NULL;
DATA[15].index = 15;
DATA[15].speed = 1;
DATA[15].speedkts = 12;
DATA[15].course = 220;
DATA[15].name = "KIROV";

WTobject_setdata(kirv,&DATA[15]);
```

```c
DATA[16].currpath = NULL;
DATA[16].speed = 1;
DATA[16].speedkts = 12;
DATA[16].course = 220;
DATA[16].name = "SLAVA";

WTobject_setdata(slv,&DATA[16]);

DATA[17].currpath = NULL;
DATA[17].speed = 1;
DATA[17].speedkts = 12;
DATA[17].course = 220;
DATA[17].name = "KRESTA I";

WTobject_setdata(kres1,&DATA[17]);

DATA[18].currpath = NULL;
DATA[18].speed = 1;
DATA[18].speedkts = 12;
DATA[18].course = 220;
DATA[18].name = "KRESTA I";

WTobject_setdata(kres2,&DATA[18]);

DATA[19].currpath = NULL;
DATA[19].speed = 1;
DATA[19].speedkts = 12;
DATA[19].course = 220;
DATA[19].name = "SOVREMMENY";

WTobject_setdata(sov1,&DATA[19]);

DATA[20].currpath = NULL;
DATA[20].speed = 1;
DATA[20].speedkts = 12;
DATA[20].course = 220;
DATA[20].name = "SOVREMMENY";

WTobject_setdata(sov2,&DATA[20]);

DATA[21].currpath = NULL;
DATA[21].speed = 1;
```

```
DATA[21].speedkts = 12;
DATA[21].course = 220;
DATA[21].name = "OILERSKI";

WTobject_setdata(ol,&DATA[21]);

DATA[22].currpath = fight1;
DATA[22].index = 22;
DATA[22].speed = 1;
DATA[22].speedkts = 550;
DATA[22].course = 180;
DATA[22].name = "F-14 TOMCAT";

WTobject_setdata(ftr1,&DATA[22]);

DATA[23].currpath = fight3;
DATA[23].index = 23;
DATA[23].speed = 1;
DATA[23].speedkts = 550;
DATA[23].course = 180;
DATA[23].name = "F-14 TOMCAT";

WTobject_setdata(ftr3,&DATA[23]);

DATA[24].currpath = fight5;
DATA[24].index = 24;
DATA[24].speed = 1;
DATA[24].speedkts = 550;
DATA[24].course = 180;
DATA[24].name = "F-14 TOMCAT";

WTobject_setdata(ftr5,&DATA[24]);

DATA[25].currpath = fight7;
DATA[25].index = 25;
DATA[25].speed = 1;
DATA[25].speedkts = 550;
DATA[25].course = 180;
DATA[25].name = "F-14 TOMCAT";

WTobject_setdata(ftr7,&DATA[25]);

DATA[26].currpath = tanker1;
```

```
DATA[26].index = 26;
DATA[26].speed = 1;
DATA[26].speedkts = 400;
DATA[26].course = 180;
DATA[26].name = "TANKER";

WTobject_setdata(tnkr,&DATA[26]);

WTobject_attach(bm1,bm3);
WTobject_attach(bm1,bm4);
WTobject_attach(bm1,bm5);
WTobject_attach(bm1,bm6);
WTobject_attach(bm1,bm7);
WTobject_attach(bm1,bm8);
WTobject_attach(bm1,bm9);
WTobject_attach(bm1,bm10);
WTobject_attach(bm1,bm11);
WTobject_attach(bm1,bm12);
WTobject_attach(bm1,bm13);

WTobject_attach(bm2,bm14);
WTobject_attach(bm2,bm15);
WTobject_attach(bm2,bm16);
WTobject_attach(bm2,bm17);
WTobject_attach(bm2,bm18);
WTobject_attach(bm2,bm19);
WTobject_attach(bm2,bm20);
WTobject_attach(bm2,bm21);
WTobject_attach(bm2,bm22);
WTobject_attach(bm2,bm23);
WTobject_attach(bm2,bm24);

WTobject_attach(ftr1,ftr2);
WTobject_attach(ftr3,ftr4);
WTobject_attach(ftr5,ftr6);
WTobject_attach(ftr7,ftr8);

curr= bm1;

Data = *(struct data*)WTobject_getdata(curr);

    spaceball=WTspaceball_new(COM1);
mouse=WTmouse_new();
```

```
                sensor=spaceball;

        window1=WTwindow_new(420,400,600,400,WTWINDOW_DEFAULT);
        WTwindow_setbgcolor(window1,0x000);
        obview= WTviewpoint_new();

        window2=WTwindow_new(0,400,400,400,WTWINDOW_DEFAULT);
        WTwindow_setbgcolor(window2,0x000);
        overview= WTviewpoint_new();

        WTviewpoint_setposition(overview,overpos);
        WTviewpoint_setorientation(overview,overornt);
        WTviewpoint_setdirection(overview,overdir);
        WTviewpoint_setviewangle(overview,WTDEFAULT_VIEWANGLE);
        WTviewpoint_sethithervalue(overview,0.05);

        WTwindow_setviewpoint(window2,overview);

                WTuniverse_setactions(actions);
                WTuniverse_ready();
                WTlight_setambient(.8);

/*UNIVERSE_GO*/
                WTuniverse_go();

/*UNIVERSE_DELETE*/
                WTuniverse_delete();
                return 0;

                }

        /* UNIVERSE ACTIONS* /
        static void actions()

{
        int dummy;

/*CALL UPDATE*/
        Update();

/*BUTTON 2*/
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON2){
```

```c
        printf("BUTTON 2 ACCEPTED \n");

/*PLAY DEFINED PATHS*/
            WTpath_setobject(bomer1,bm1);
            WTpath_setplayspeed(bomer1,5);
            WTpath_play(bomer1);
            WTpath_setobject(bomer2,bm2);
            WTpath_setplayspeed(bomer2,5);
            WTpath_play(bomer2);
            WTpath_setobject(fight1,ftr1);
            WTpath_setplayspeed(fight1,5);
            WTpath_play(fight1);
            WTpath_setobject(fight3,ftr3);
            WTpath_setplayspeed(fight3,5);
            WTpath_play(fight3);
            WTpath_setobject(fight5,ftr5);
            WTpath_setplayspeed(fight5,5);
            WTpath_play(fight5);
            WTpath_setobject(fight7,ftr7);
            WTpath_setplayspeed(fight7,5);
            WTpath_play(fight7);
            WTpath_setobject(tanker1,tnkr);
            WTpath_setplayspeed(tanker1,5);
            WTpath_play(tanker1);
            WTpath_setobject(helo1,h1);
            WTpath_setplayspeed(helo1,5);
            WTpath_play(helo1);
            WTpath_setobject(helo2,h2);
            WTpath_setplayspeed(helo2,5);
            WTpath_play(helo2);
            WTpath_setobject(helo3,h3);
            WTpath_setplayspeed(helo3,5);
            WTpath_play(helo3);
        }
/*BUTTON 3*/
        if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON3){

        printf("BUTTON 3 ACCEPTED \n");

        curr= WTuniverse_pickobject(
            *(WTp2*)WTsensor_getrawdata(mouse));
```

```c
        Data = *(struct data*)WTobject_getdata(curr);

/*CALL UPDATE*/
        Update();

/*PRINT DATA*/
        printf("The platform is %s \n",Data.name);
        printf("The course is %d \n",Data.course);
        printf("The speed is %d knots \n",Data.speedkts);
    }
/*BUTTON 4*/
    if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON4){

        printf("BUTTON 4 ACCEPTED \n");

/* OBJECT WINDOW VIEW EQUALS UNIVERSE WINDOW VIEW - indirectly,
because Data.currpath = NULL views will be the same. */

        curr= WTuniverse_pickobject(
            *(WTp2*)WTsensor_getrawdata(mouse));

        Data.currpath = NULL;


    }
/*BUTTON 6*/
    if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON6){

        printf("BUTTON 6 ACCEPTED \n");

        curr= WTuniverse_pickobject(
            *(WTp2*)WTsensor_getrawdata(mouse));

        Data = *(struct data*)WTobject_getdata(curr);

/*CALL UPDATE*/
        Update();

/*PRINT DATA*/
        printf("The platform is %s \n",Data.name);
        printf("The course is %d \n",Data.course);
        printf("The speed is %d knots \n",Data.speedkts);

/*CALL CHANGE PATH*/
```

```
        if (Data.currpath != NULL)
         {
           printf("Do you want to change the path (Y/N)?\n");

              dummy = getchar();
                getchar();

              switch (dummy)
              {
                 case('Y') : {
                                Change_path();
                       break;}

                     default : {
                                  break;}
              }
           WTpath_setobject(Data.currpath,curr);
           WTpath_play(Data.currpath);

         }
         }

/*BUTTON 7*/
         if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON7){

           printf("BUTTON 7 ACCEPTED \n");

/*ATTACH SPACEBALL TO UNIVERSE VIEW*/
             WTsensor_setsensitivity(sensor, 0.1 *WTuniverse_getradius());
         WTviewpoint_addsensor(view,spaceball);
          }

  /*BUTTON 1*/
         if(WTsensor_getmiscdata(spaceball)&WTSPACEBALL_BUTTON1){

           printf("BUTTON 1 ACCEPTED \n");

                WTuniverse_stop();
          }
}

/*UPDATE()*/
void Update()
```

```
{
/*UPDATE UNIVERSE VIEW*/
        float u, v;

        WTp3 obdir,obpos,obpos1,nodepos1,nodepos2,adjust,cockpit;
        WTq obornt;
        WTp3_init(cockpit);

        view= WTuniverse_getviewpoint();

        if (Data.currpath != NULL)
        {
         currnode= WTpath_getcurrentnode(Data.currpath);
         WTpathnode_getposition(currnode,obpos1);
         WTpath_seek(Data.currpath,Data.speed,WTPATH_CURRENT);
         nextnode= WTpath_getcurrentnode(Data.currpath);
         WTpathnode_getposition(nextnode,obpos);
         WTpathnode_getorientation(nextnode,obornt);
         WTpath_seek(Data.currpath,-Data.speed,WTPATH_CURRENT);

         WTp3_subtract(obpos,obpos1,obdir);
         WTobject_alignaxis(curr,Z,obdir);

         u = obdir[X];
         v = obdir[Z];

         DATA[Data.index].course = Course_read(&u,&v);
         Data.course = DATA[Data.index].course;

         WTp3_add(cockpit,obdir,cockpit);

         WTp3_mults(cockpit,4);

         WTp3_add(obpos,cockpit,obpos);

         WTviewpoint_setposition(obview,obpos);
         WTviewpoint_setorientation(obview,obornt);
         WTviewpoint_setdirection(obview,obdir);
         WTviewpoint_setviewangle(obview,WTDEFAULT_VIEWANGLE);
         WTviewpoint_sethithervalue(obview,0.05);
        }
        else
```

```
        {
        WTobject_getposition(curr,obpos);
        WTobject_getorientation(curr,obornt);
        WTobject_getaxis(curr,Z,obdir);

        WTviewpoint_setposition(obview,obpos);
        WTviewpoint_setorientation(obview,obornt);
        WTviewpoint_setdirection(obview,obdir);
        WTviewpoint_setviewangle(obview,WTDEFAULT_VIEWANGLE);
        WTviewpoint_sethithervalue(obview,0.05);
        }

        WTwindow_setviewpoint(window1,obview);

/*RETURN*/
return;
}

/*CHANGE_PATH()*/
void Change_path()
 {

/*ALLOW USER TO CHANGE PATH*/
int pathoption;

Again:
printf("To create a new path, type 'N'\n");
printf("To quit, type 'Q'\n");

pathoption = getchar();
getchar();

switch (pathoption)
{
        case('N') : {
                        New_path();
                        break;}

        case('Q') : {
                        Quit_();
                        break;}

        default: {
```

```
                    printf("Try again ... \n");
                    goto Again;}

}

/*RETURN*/
return;

}

void New_path()
{

        WTpq pq;
           int dummy;
           int point_num = 1;
           int coord, node_x_pos, node_y_pos, node_z_pos;
           int node_x_euler, node_y_euler, node_z_euler;
        float distance = 0;
        float point_dist = 0;
        WTp3 last;
        float spd;
        WTp2 mousepos;

        currnode= WTpath_getcurrentnode(Data.currpath);

        tempnode = WTpathnode_copy(currnode);

        tempname = WTpath_new(nodeobj);

        WTpath_appendnode(tempname,tempnode);

        WTpathnode_getposition(tempnode,last);

Another_point:
        point_num++;


NODE:
            printf("Do you want input with the mouse (Y/N)?\n");

                dummy = getchar();
                   getchar();
```

```
                    switch (dummy)
                    {
                        case('Y') : {
                                    goto MOUSE;}

                        default : {
                            goto KEYBRD;}
                    }
MOUSE:
            WTsensor_setmiscdata(mouse,NULL);

            WTmouse_rawupdate(mouse);
            raw = (WTmouse_rawdata *)WTsensor_getrawdata(mouse);

        WTsensor_setmiscdata(mouse,NULL);

        while (!WTsensor_getmiscdata(mouse)&WTMOUSE_LEFTBUTTON)
        {
          WTmouse_rawupdate(mouse);
          raw = (WTmouse_rawdata *)WTsensor_getrawdata(mouse);

        }

spd = WTsensor_getsensitivity(mouse);

pq.p[X] = (raw->pos[X] - (WTwidth/2)) * spd / (WTwidth/2);
pq.p[Z] = ((WTheight/2) - raw->pos[Y]) * spd / (WTheight/2);

pq.p[X] = (pq.p[X] / .005566) + 24.5;
pq.p[Z] = (pq.p[Z] / -0.008833) - 291.5;


printf("Input an altitude \n");

pq.p[Y] = Get_coord();

WTq_copy(overornt,pq.q);

WTp3_print(pq.p,"pq.p = \n");

goto BACK;
```

```
KEYBRD:
        printf("point # %u\n",point_num);

        printf("X coordinate?  ");
        node_x_pos = Get_coord();
        printf("Y coordinate?  ");
        node_y_pos = Get_coord();
        printf("Z coordinate?  ");
        node_z_pos = Get_coord();

printf("node x position=   %d\n",node_x_pos);
printf("node y position =   %d\n",node_y_pos);
printf("node z position =   %d\n",node_z_pos);

pq.p[X] = node_x_pos;
pq.p[Y] = node_y_pos;
pq.p[Z] = node_z_pos;

        printf("X euler orientation (degrees)?  ");
        node_x_euler = Get_coord();
        printf("Y euler orientation (degrees)?  ");
        node_y_euler = Get_coord();
        printf("Z euler orientation (degrees)?  ");
        node_z_euler = Get_coord();

printf("node x orientaion =   %d\n",node_x_euler);
printf("node y orientaion =   %d\n",node_y_euler);
printf("node z orientaion =   %d\n",node_z_euler);

node_x_euler = node_x_euler * PI / 180;
node_y_euler = node_y_euler * PI / 180;
node_z_euler = node_z_euler * PI / 180;

WTeuler_2q(node_x_euler,node_y_euler,node_z_euler,pq.q);


BACK:

printf("Is this node correct, type 'Y'\n");

dummy = getchar();
getchar();
```

```c
    switch (dummy)
    {
            case('Y') : {
                        break;}


            default: {
                        printf("Try again ... \n");
                        goto NODE;}

    }
    point_dist = WTp3_distance(pq.p,last);
    printf("point_dist = %f \n",point_dist);
    distance = distance + point_dist;
    printf("distance = %f \n",distance);
    WTp3_print(last,"last = \n");
    WTp3_print(pq.p,"pq.p = \n");

    tempnode = WTpathnode_new(&pq);

    WTpath_appendnode(tempname,tempnode);

printf("Do you want to input another point (Y/N)?\n");

dummy = getchar();
getchar();

        switch (dummy)
            {
                case('N') : {
                            Interpolate(&distance);
                        return;}

                default : {
                    WTp3_copy(pq.p,last);
                            goto Another_point;}
            }

}

void Quit_()
{
```

```c
        printf("stop\n");
        return;
}
int Get_coord()
{
int coord;
char inputStr[max];
getInStr(inputStr,max);
coord = atoi(inputStr);
return (coord);
}

void getInStr(char str[],int len)
{
        int i = 0, inputChar;
        inputChar = getchar();
        while (i<(len-1) && (inputChar!='\n'))
        {
                str[i] = inputChar;
                i++;
                inputChar = getchar();
        }
        str[i]='\0';
        return;
}

void Interpolate(float *distance)
{
  int dummy,points,coord,i,j;
  float frac_points;
  WTp3 last;

frac_points = *distance / (oneknot * Data.speedkts);
points = ceil(frac_points);

method:
printf("For Linear interpolation, type 'L'\n");
printf("For Bezier interpolation, type 'B'\n");
printf("For B-spline interpolation, type 'S'\n");

dummy = getchar();
getchar();
```

```c
switch (dummy)
{
        case('L') : {

                tempname =
WTpath_interpolate(tempname,points,WTPATH_LINEAR);
                break;}
        case('B') : {
                tempname =
WTpath_interpolate(tempname,points,WTPATH_BEZIER);
                break;}
        case('S') : {
                tempname =
WTpath_interpolate(tempname,points,WTPATH_BSPLINE);
                break;}

        default: {
                printf("Try again ...\n");
                goto method;}

}
     WTpath_delete(Data.currpath);
     Data.currpath = WTpath_copy(tempname);
     WTpath_rewind(Data.currpath);

return;}


int Course_read(float *u, float *v)
{
double ratio;
int course,ref,sgn,intangle;
float angle, x, z;

x = *u;
z = *v;

if ((x==0)||(z==0))
{
        if ((x==0)&&(z>=0))
        { course = 0;}
    if ((x==0)&&(z<0))
     { course = 180;}
```

```
        if ((z==0)&&(x>=0))
            { course = 90;}
        if ((z==0)&&(x<0))
        { course = 270;}
}
else
{       ratio = (z/x);
        angle = sqrt(atan(ratio) * atan(ratio));
        intangle = (angle * 180.0) / PI;


    if (x>0)
    {
        if (z>0)
        { ref = 90;
        sgn = -1;}
      else
      {  ref = 90;
        sgn = 1;}
    }
    if (x<0)
    {
        if (z>0)
        { ref = 270;
        sgn = 1;}
      else
      { ref = 270;
        sgn = -1;}
    }

course = ref + (sgn * intangle);

}
return(course);}
```

LIST OF REFERENCES

1. Pimentel, K. and Teixeira, K., *Virtual Reality, Through the New Looking Glass*, p. xv, Intel/Windcrest/Mcgraw-Hill Inc., 1992.

2. Sense8 Corp., *World Tool Kit Reference Manual*, p. 2-8, Sense8 Corp., 1993.

BIBLIOGRAPHY

1. Pimentel, K. and Teixeira, K., *Virtual Reality, Through the New Looking Glass*, Intel/Windcrest/Mcgraw-Hill Inc., 1992.

2. Sense8 Corp., *World Tool Kit Reference Manual*, Sense8 Corp., 1993.

3. Perry, G., *C by Example*, Que 1993.

4. Jamsa, K., *Jamsa's 1001 C/C++ Tips*, Jamsa Press, 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Superintendent 2
   Attn: Library, Code 52
   Naval Postgraduate School
   Monterey, California 93943-5000

3. Prof. Paul Moose 4
   Attn: C3 Joint Academic Group, Code CC
   Naval Postgraduate School
   Monterey, California 93943-5000

4. Prof. Morris R. Driels 3
   Attn: Mechanical Engineering Dept., Code ME
   Naval Postgraduate School
   Monterey, California 93943-5000

5. Lt. John M. Young 2
   5944 Cleary Rd.
   Livonia, New York 14487